RICE UNIVERSITY

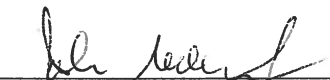
Software Support for Efficient Use of Modern Computer Architectures

by

Milind Chabbi

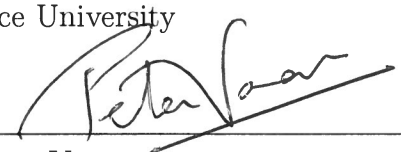
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree
Doctor of Philosophy

Approved, Thesis Committee:



John Mellor-Crummey
Professor of Computer Science and
Electrical and Computer Engineering
Rice University

Vivek Sarkar
Professor and Chair,
Department of Computer Science
and E.D. Butcher Chair in Engineering
Rice University



Peter Varman
Professor of Electrical and Computer
Engineering
Rice University

Costin Iancu
Staff Scientist
Lawrence Berkeley National Laboratory

Houston, Texas
July, 6, 2015

ABSTRACT

Software Support for Efficient Use
of Modern Computer Architectures

by

Milind Chabbi

Parallelism is ubiquitous in modern computer architectures. *Heterogeneity* of CPU cores and *deep memory hierarchies* make modern architectures difficult to program efficiently. Achieving top performance on supercomputers is difficult due to complex hardware, software, and their interactions.

Production software systems fail to achieve top performance on modern architectures broadly due to three main causes: *resource idleness*, *parallel overhead*, and *data movement overhead*. This dissertation presents novel and effective *performance analysis tools*, *adaptive runtime systems*, and *architecture-aware algorithms* to understand and address these problems.

Many future high performance systems will employ traditional multicore CPUs augmented with accelerators such as GPUs. One of the biggest concerns for accelerated systems is how to make best use of both CPU and GPU resources. *Resource idleness* arises in a parallel program due to insufficient parallelism and load imbalance, among other causes. To assess systemic resource idleness arising in GPU-accelerated architectures, we developed efficient profiling and tracing capabilities. We introduce CPU-GPU blame shifting—a novel technique to pinpoint and quantify the causes of resource idleness in GPU-accelerated architectures.

Parallel overheads arise due to synchronization constructs such as barriers and locks used

in parallel programs. We developed a new technique to identify and eliminate redundant barriers at runtime in Partitioned Global Address Space programs. In addition, we developed a set of novel mutual exclusion algorithms that exploit locality in the memory hierarchy to improve performance on Non-Uniform Memory Access architectures.

In modern architectures, *inefficient or unnecessary memory accesses* can severely degrade program performance. To pinpoint and quantify wasteful memory operations, we developed a fine-grain execution-monitoring framework. We extended this framework and demonstrated the feasibility of attributing fine-grain execution metrics to source and data in their contexts for long running programs—a task previously thought to be infeasible.

The solutions described in this dissertation were employed to gain insights into the performance of a collection of important programs—both parallel and serial. The insights we gained enabled us to improve the performance of several important programs by a significant margin. Software for future systems will benefit from the techniques described in this dissertation.

Acknowledgments

I wish to thank my thesis committee: Prof. John Mellor-Crummey, Prof. Vivek Sarkar, Prof. Peter Varman, and Dr. Costin Iancu.

It is exceptionally difficult to acknowledge the help of my advisor Prof. John Mellor-Crummey, which has extended beyond the time I spent at Rice University and beyond the academic realm. Prof. Mellor-Crummey's paper about the MCS lock, which I read in 2006 during a course under Prof. Greg Andrews at The University of Arizona, sparked my interest in shared-memory synchronization. It was the same year the paper won the prestigious Edsger W. Dijkstra Prize in Distributed Computing. I had my first interaction with Prof. Mellor-Crummey in the Fall of 2008, when I emailed him with my interest in his work. Prof. Mellor-Crummey's prompt reply and the ongoing PACE project encouraged me to consider Rice seriously. Although I was admitted to Rice in 2009, I could not join that year due to immigration complications. Prof. Mellor-Crummey was patient to wait until I joined Rice in the Fall of 2010. In the months and years that followed, I would only see Prof. Mellor-Crummey's abilities with awe: how he pulled a reference to a related work in matter of seconds and how he remembered author's names, paper titles, and processor architectures.

Early years of the graduate studies were filled with many "fake eureka" moments, which I would share with Prof. Mellor-Crummey, he would patiently listen to my "ideas" of reinventing the wheel, and respectfully point me to a work in 1990's that had already solved it, in a better way on many occasions. As the time progressed, some seeds of ideas grew and started showing promise. Over the years Prof. Mellor-Crummey has helped me set higher standards in my research goals and impact. He helped me turn my half-baked ideas into concrete solutions. He would ask me the right set of questions, in search of answers through which I would learn a great deal about my own work, improve my ideas, and know more about the state-of-the-art. In stark contrast with my experience in industry, where doing anything outside the defined boundaries was a taboo, working under Prof. Mellor-Crummey,

there is no research question that is not worth pursuing. I am fortunate to have explored compilers, heterogenous architectures, high performance computing, binary analysis, performance analysis, data race detection, model checking, synchronization, computer security, and several other aspects in my graduate career.

Having started my graduate career with inspiration from the MCS lock, it was my desire to contribute in that direction. We came close to doing something related to locks multiple times during the last three years: binary analysis for identifying mutual exclusion on one occasion and building efficient reader-writer locks on two occasions. However, we had to give up both because it was too hard to do or already been done very recently. The insight came when Prof. Mellor-Crummey and I were looking at lock contention in NWChem. Prof. Mellor-Crummey came up with the idea of batching requests from the same node when there were local contenders; I was already aware of lock cohorting. After that we both envisioned an MCS lock with multiple levels of hierarchy to take advantage of NUMA machines. Subsequently, we devised the HMCS lock, showed its superiority both theoretically and empirically. This work has only improved over time with new design for an adaptive HMCS lock and more. I am indebted to Prof. Mellor-Crummey for always holding the bar higher for me to reach newer heights.

Beyond research, Prof. Mellor-Crummey has helped me in my presentation skills, connected me with several researchers, and helped plan my vacations too.

I would like to thank the HPCTOOLKIT research staff: Laksono Adhianto, Mark Krentel, and Mike Fagan. Without their constant support this work would not have been possible. I would like to call out Mike Fagan for collaborating in many of my projects and being extremely helpful in paper writing and solving both computer science and mathematical problems.

I want to thank my collaborators Costin Iancu, Wim Lavrijsen, and Wibe de Jong from Lawrence Berkeley National Laboratory. I am indebted to Costin for his warmth, connecting me with people, and helping with my job search.

I would like to thank Dr. Tracy Volz at Rice University for training me on many occasions for my presentations and providing me with the opportunity to train other students at Rice.

This acknowledgement will be incomplete without mentioning the great support form

the administrative staff of the department of computer science at Rice University. I would like to particularly thank Belia Martinez for her warmth and friendliness, which make her the friend of all graduate students. Belia also deserves accolades for helping me with my immigration process.

This acknowledgement will be incomplete without mentioning the constant feedback that fellow graduate students provided in my talks, research work, and publications, including this dissertation.

Reshmy, Rajesh, Aarthi, Shruti, Priyanka, Deepak, Karthik, Rahul, Gaurav, and Rajoshi, who became my Rice family had a positive role to play. They made my years in Houston at Rice memorable and fun filled.

I would like to thank Girish for being a close friend in Houston. Few people would shamelessly eat dinner at a friend's place as many times as I have eaten in Girish's house.

A large round of thanks to my father, mother, and brother. They have stood with me through my good and bad times. They have encouraged me to accomplish my dreams, shown confidence in my aspirations, and provided moral, spiritual, and financial support at all times. I would like to thank my wife for her unconditional support in my endeavors.

I wish to thank the funding agencies that supported my work: Defense Advanced Research Projects Agency (DARPA), U.S. Department of Energy (DOE), Texas Instruments, Ken Kennedy Institute for Information Technology, and Lawrence Berkeley National Laboratory. I am thankful to the world class computational facilities provided by XSEDE, U.S. DOE laboratories, and Rice University's Research Computing Support Group, which made this research possible.

Specifically, this work is funded in part by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915, DOE Office of Science under Cooperative Agreements DE-FC02-07ER25800, DE-FC02-13ER26144, DE-FC02-12ER26119, DE-SC0008699, and DE-SC0010473, Sandia National Laboratory purchase order 1293383, and Lawrence Berkeley National Laboratory subcontract 7106218.

This research used the resources provided by the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory supported by the DOE Office of Science under Contract No. DE-AC05-00OR22725, Keeneland Computing Facility at the Georgia

Institute of Technology supported by the National Science Foundation under Contract OCI-0910735, Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097, Blacklight system at the Pittsburgh Supercomputing Center (PSC) via the Extreme Science and Engineering Discovery Environment (XSEDE) supported by National Science Foundation grant number OCI-1053575, and IBM Shared University Research (SUR) Award in partnership with CISCO, Qlogic and Adaptive Computing acquired with support from NIH award NCRR S10RR02950.

Contents

Illustrations

Tables

Chapter 1

Introduction

> *Don't lower your expectations to meet your performance. Raise your level of performance to meet your expectations.*
>
> Ralph Marston

There is an ever-widening chasm between the peak performance of microprocessor-based systems and the actual performance achieved by an application software. The peak performance of a microprocessor, to this day, is governed by the Moore's Law [160] (more properly called Moore's Conjecture). Moore's Law predicts the number of transistors in an integrated circuit to double every two years. The performance realized by typical scientific applications on microprocessor-based systems is, however, abysmally low—only 5-15% of the peak [179].

Computer architectures are increasing in complexity. Modern microprocessor architectures have evolved from single-threaded, single-core processors into multi-threaded, many-core architectures. Furthermore, many modern computers are a heterogeneous combination of a few *latency-optimized* heavyweight CPU cores, augmented with *throughput-optimized* lightweight cores, a.k.a. accelerators [100, 101, 104, 113, 176]. Such designs are favored over traditional homogeneous multicore designs since accelerators deliver high throughput at a modest energy budget [1, 70, 92, 113, 174, 214]. In addition, modern computer architectures are further complicated by deep memory hierarchies [16] comprised of multiple levels of cache and memory, distributed over multiple sockets, and in some systems, multiple nodes [208]. These deep memory hierarchies lead to different memory access latencies for data residing in different memory locations—commonly known as Non-Uniform Memory Access (NUMA). Finally, the architectures change rapidly, making hardware obsolete often faster than an automatic, compiler-driven optimization can hope to target particular hardware.

Large production software systems are complicated as well. To ease the understanding of complicated systems, software developers traditionally employ layers of abstractions. Back in the day, software was written for a single machine. Over the years, strong boundaries have been drawn between hardware, operating system, and application software. Abstractions such as routines and libraries of routines ensure separation of concerns and provide reuse of components. Software abstractions, however, come at a cost—they may introduce overheads by causing redundancies and hindering optimizations. Rapid hardware evolution makes it harder for software to adapt or take advantage of modern architectural features. Legacy layered software, often written to be architecture agnostic, fails to deliver top performance on current hardware.

1.1 Performance challenges

Performance losses occur due to various causes at multiple layers of the software stack. There is a litany of reasons for performance losses. Some of the most common ones are: insufficient parallelism, load imbalance, use of heavyweight abstractions, developers' inattention to performance, poor choice of algorithms and data structures, deleterious compiler optimizations, layering of software stack leading to overheads unknown to an application developer, and mismatch between hardware and software characteristics.

Until recently, inefficiencies in software were mitigated by a rapid increase in CPU clock rates along with hardware advancements such as speculation and instruction-level parallelism. With the end of Dennard scaling [24], the increase in CPU clock rates has stalled. Superscalar architectural innovations appear to have run their course [180]. Applications have limited instruction-level parallelism. Speculation in hardware is useful but is not a panacea. Current day hardware innovation is mostly coming from rising CPU core counts. Since power consumption is a critical limiting factor in scaling hardware [214], modern hardware often employs simpler cores. These simpler cores run at slower clock speeds. Simple cores also have less advanced features, e.g., no out-of-order execution and no hardware speculation. In this era of exploding parallelism and less powerful individual cores, tuning applications to take advantage of hardware features mandates tuning applications at all levels—both single-threaded and multi-threaded, both inter-node and intra-node.

1.1.1 Insufficiency of state-of-the-art performance analysis techniques

Performance analysis tools play a vital role in pinpointing and quantifying performance bottlenecks in large software systems. Effective performance analysis tools deliver insights that empower a developer to improve an application's performance and take advantage of modern architectures. Moreover, good performance analysis tools are crucial for tuning systems where automatic optimizations are infeasible.

Hot spot analysis is a classical technique for identifying performance problems in an execution. Hot spot analysis informs the developer about code regions that consume large amounts of resources such as CPU cycles. Intel VTune [98], a well-known performance analysis tool, defines hot spot analysis as follows: *"Hot spot analysis helps understand application flow and identify sections of code that get a lot of execution time (hot spots). A large number of samples collected at a specific process, thread, or module can imply high processor utilization and potential performance bottlenecks. Some hot spots can be removed, while other hot spots are fundamental to the application functionality and cannot be removed."*

Hot spot analysis focuses only on resources *being used*. In modern hardware with multiple components, such as vector execution units, hardware multi-threading, multiple CPU cores, and accelerator cores, some (in fact many) resources remain idle. Hot spot analysis cannot quantify the losses arising from resources *not being used*. Furthermore, hot spot analysis typically measures metrics such as floating-point operations per second (FLOPS) or cycles per instruction (CPI). A low CPI (or high FLOPS) ensures the application is not stalled for resources; however, it does not ensure the application is making *good use* of its resources. Execution hot spots can only point to the "symptoms" of performance problems at best.

This dissertation approaches performance analysis of parallel programs in a top-down manner in the order-of-magnitude of causes. First order performance losses happen in parallel programs due to *resource idleness* caused by improper work partitioning and serialization, among others. Second order performance losses happen in parallel programs due to *parallel overheads* caused by synchronization constructs such as locks and barriers. Third order performance losses happen due to *memory access overheads* within individual threads of executions. This ordering provides a mental model to approach performance analysis and

tuning. In fact, the aforementioned factors can be interrelated. For example, synchronization overheads can cause idleness; memory access behavior can depend on work decomposition.

Parallel overheads can be due to *unnecessary* or *inefficient* synchronization—this dissertation examines instances of both. Memory access overheads can also be *unnecessary* or *inefficient*. In fact, the performance impact of inefficient memory accesses can be more severe than that of parallel overheads. Inefficient memory accesses have been extensively studied elsewhere [118, 133, 134, 135, 136, 138, 156, 237]. This dissertation examines only *unnecessary* memory accesses; and hence, we place it third in our list of tuning priorities.

Effective performance tools should not only measure performance efficiently but also attribute the metrics to source code and data objects in context, accurately. Contextual performance attribution helps in better understanding of a program's behavior. Contextual attribution also enables specific optimizations possible only in certain execution contexts. As the final aspect of performance analysis, this dissertation considers the *contextual attribution of performance*.

In Section 1.1.2-1.1.5, we provide the background necessary to motivate the problem of *resource idleness*, *parallel overhead*, *memory access overhead*, and the *need for contextual attribution*. We develop solutions to address these problems in the subsequent chapters.

1.1.2 Underutilization of resources in parallel programs

The overall execution time of a parallel application is governed by its critical path length [239]—the longest execution sequence without wait states. Clearly, short critical path is best. Load imbalance causes many resources to idle waiting for a few (sometimes just one) working processes to finish along the *critical path*. A key source of inefficiency in parallel programs is resource idleness. Shortening the parts of the critical path where several resources are idle is key to better resource utilization. We mention in passing that emerging work scheduling runtimes [37, 50, 73, 78] offer a rich set alternatives for better resources utilization for program regions where the amount of parallelism exceeds the available resources (parallel slackness). Current implementations of work scheduling schemes, however, typically have a tight coupling with a language or a platform, which may incur higher overheads. Furthermore, legacy HPC applications, which typically employ multiple layers of libraries, program-

ming languages, and programming models, are harder to retarget to these work scheduling runtimes.

Identifying the critical path in an execution is a key aspect in tuning parallel codes. There are many prior efforts in tuning applications by identifying critical paths [23, 28, 51, 239], but they rely on either heavyweight execution trace collection or simulation. Instrumentation for execution trace collection can distort the true nature of the critical path. Another common alternative, simulating a parallel execution, is fundamentally non-scalable. Rice University's HPCTOOLKIT has already addressed idleness analysis with lightweight profiling in homogeneous many-core CPU environments [137, 222, 223, 225].

Idleness is a major source of inefficiencies in emerging heterogeneous architectures that employ CPUs and GPUs. No prior efforts, however, have been made to assess idleness in the heterogeneous CPU-GPU execution paradigm. Figure 1.1 is an execution trace of the GPU accelerated version of LAMMPS [189]—a key molecular dynamics code. The x-axis represents the time dimension. The y-axis represents activities on various compute resources. The top row represents the CPU execution. The bottom two rows represent two concurrent streams executing on a GPU. The gray color in the trace indicates idleness. Other colors represent different functions being executed and imply useful work. The total utilization of the GPU is the aggregated utilization of both streams.

In this code, the CPU and GPUs are used in a mutually exclusive fashion. As a result, the execution exhibits a vast amount of resource idleness—both on the CPU and on the GPU. Improper work partitioning, CPU throttling, operating system blocking, delays in CPU-GPU communication, among others contribute to load imbalance and lead to idleness in heterogeneous systems. All prior efforts in GPU performance analysis have focused solely on identifying resources being excessively used, in line with the classical hot spot analysis technique. To address this deficiency in state-of-the-art performance analysis, we need efficient profiling and tracing techniques that guide a developer to better understand hybrid code executions and easily identify code regions best suited for tuning.

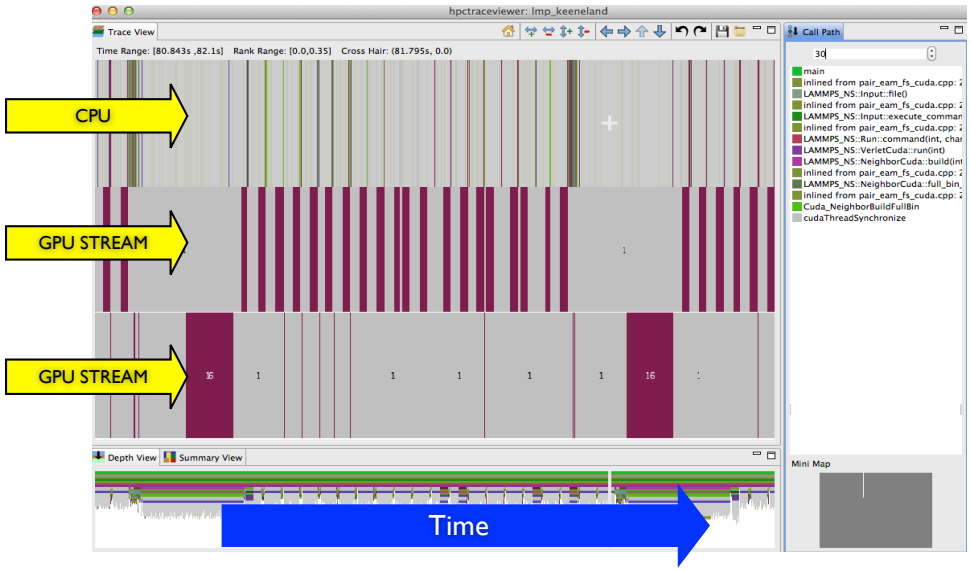


Figure 1.1 : Execution trace of LAMMPS-CUDA code running on the Keeneland GPU compute cluster [232]. Gray color in GPU stream indicates no work on the GPU streams. Gray color in CPU indicates CPU idle waiting for GPU to finish. CPU and GPU executions are not overlapped in time.

1.1.3 Synchronization overhead in parallel programs

Few real-world problems are embarrassingly parallel. Many real-world problems require coordination and communication between parallel components. Synchronization by means of *barriers* is essential to separate programs into different phases. Synchronization by means of *locks* is equally necessary to ensure mutually exclusive access to a shared data.

Once serialization and load imbalance problems are identified and addressed in a parallel program, the next cause of performance losses to consider is parallel overheads. Parallel overheads are detrimental to the scalability of an application. Synchronization constructs such as locks and barriers introduce parallel overheads. Parallel programs incur two kinds of synchronization penalties that affect parallel performance. They are:

Unnecessary synchronization: Large production software systems with several layers of abstractions make conservative assumptions about lower layers of abstractions. In such situations, synchronization in the form of fences and barriers are enforced on entry and exit to each layer of the abstraction. Global barrier synchronization is expensive and

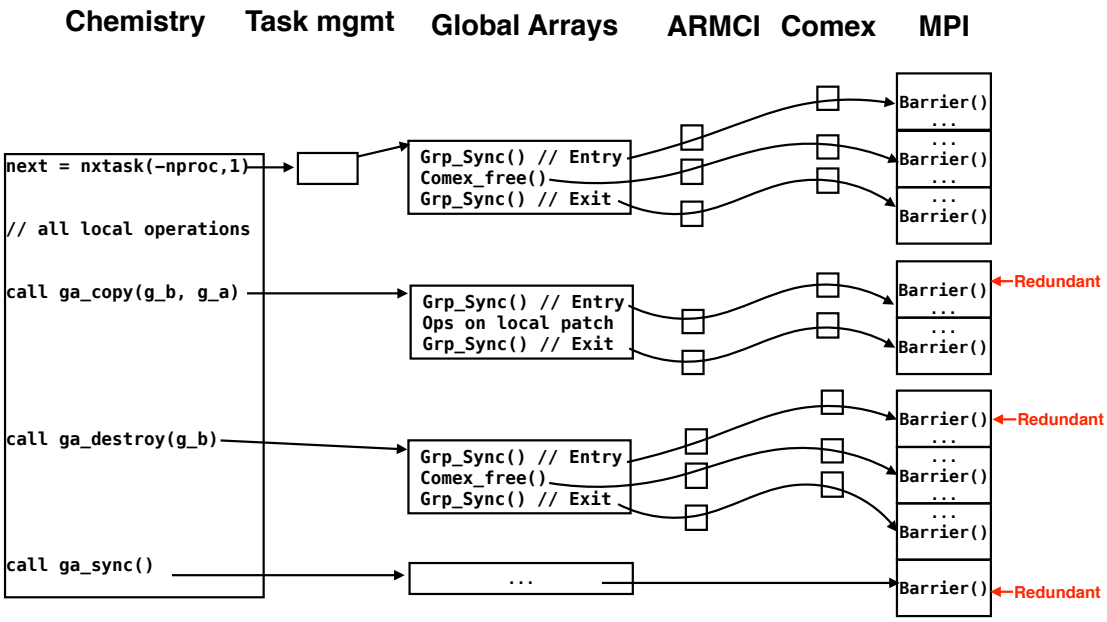


Figure 1.2 : An example of wasteful barrier synchronization in NWChem. Software layering and API composition cause back-to-back barriers without intervening updates to shared data. Three out of nine barriers are redundant.

scales poorly. Excessive synchronization, when it is not necessary, costs dearly to the performance of a parallel program. Figure 1.2 shows an example of software layering and API composition in NWChem [230]—a flagship computational quantum chemistry code—where nine barriers are executed in four lines of the application code, of which three barriers are redundant. To this day, few efforts have been made to pinpoint and quantify the cost of unnecessary synchronization in production parallel programs.

Costly synchronization: The algorithm employed to implement a synchronization primitive governs its runtime efficiency. Wrong assumptions made by a synchronization algorithm about the characteristics of an underlying hardware can cause performance losses. For example, state-of-the-art locking mechanisms developed in the 1990s assumed a flat memory hierarchy whereas modern multi-node multi-socket many-core NUMA machines have a deep distributed memory hierarchy. Due to this mismatch between algorithms and modern hardware characteristics, codes with highly contended locks fail to deliver high throughput on NUMA machines.

We need a scalable and lightweight mechanism to pinpoint redundant barriers in production parallel programs. In complex, layered software, one may not be able to eliminate redundant synchronization by simple code refactoring; compiler-driven techniques also become infeasible. For layered software systems, runtime introspection is often necessary to eliminate redundant synchronization. In addition, the changing hardware landscape demands revisiting locking algorithms designed for the previous generation of machines.

1.1.4 Memory accesses overhead in a thread of execution

For many programs, exposed memory latency accounts for a significant fraction of the execution time. Memory access overhead can be classified into two categories 1) *inefficient* memory accesses and 2) *unnecessary* memory accesses. A vast amount of literature has already been dedicated to understanding and addressing *inefficient* memory accesses, e.g., [118, 133, 134, 135, 136, 138, 156, 237]. *Unnecessary* memory accesses cause *wasteful resource consumption*. In fact, unnecessary memory accesses, such as *dead writes*, are a common source of performance problems. Causes for dead writes can be attributed to the choice of algorithms and data structures, compiler optimizations, the overhead of abstractions, and architectural characteristics, among others. The compiler literature is full of optimizing computations and memory accesses. Procedure boundaries, compilation units, aliasing, and aggregate variables, among others, hamper compiler optimizations. Similarly, compilers may not eliminate context or flow-sensitive redundancies.

Developers need access to efficient tools that can pinpoint and quantify wasteful resource consumption at a fine-grained instruction-level. Few performance analysis tools have focused on identifying wasteful resource consumption caused by unwanted computations or memory accesses in an entire execution. Space and time overheads as well as imprecision in binary analysis, pose formidable challenges to instruction-level profiling executions of long-running production codes.

1.1.5 Performance variation of the same code in different contexts

The performance of a code fragment is often context dependent. Large software applications that employ several layers of libraries need performance attributed to their contexts to

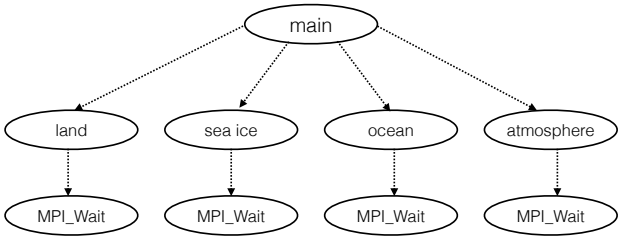


Figure 1.3 : A skeleton climate code. The wait routine can be called from various components of the application. Performance attributed to the source along with its runtime calling context is needed to pinpoint and understand problematic codes.

understand the code behaviors in different execution contexts. Furthermore, polymorphic codes, such as C++ templates, demand contextual invocation information to decipher their instantiation and runtime behavior. Figure 1.3 shows a skeleton climate code where different components of the application call the `MPI_Wait` routine. Precise calling context that leads to a call to `MPI_Wait` is needed to understand the context under which excessive idle waiting is happening. Contextual attribution of performance metrics is critical both for providing insightful feedback as well as for enabling valuable optimizations.

The performance of a code fragment not only depends on the context in which it executes but also on the data objects referenced. The same code may exhibit different performance traits based on the location and organization of the data objects that it accesses.

A recurring theme in this dissertation is attributing metrics to source contexts and data. Prior work has successfully addressed the contextual attribution of performance to code and data objects for *sampling-based coarse-grained performance analysis tools* [133, 224]. Performance attribution to code and data for *instrumentation-based fine-grained performance analysis tools*, however, was left unaddressed. In fact, attributing performance to code for every executed instruction was considered infeasible for long running programs due to purported space and time overheads [69, 229].

Attributing metrics to code and data in context is useful not only in performance analysis tools but also in tools for correctness and debugging. Software developers and researchers need an open-source framework for attributing execution characteristics to calling context and data in fine-grain monitoring tools.

1.2 Thesis statement

Production software systems suffer from performance losses at multiple layers of the software stack, primarily due to resource idleness, synchronization overhead, and memory access overhead. As a result, software does not achieve top performance on modern hardware architectures. Performance analysis tools that identify resource idleness, unnecessary synchronization, and wasteful memory accesses, provide valuable insights for application tuning. Performance-aware adaptive runtimes can eliminate unnecessary synchronization and enhance the efficiency of necessary synchronization.

1.3 Contributions

It would be impossible in a single dissertation to offer a comprehensive treatment of all aspects of performance analysis of software. Consequently, this dissertation offers methods to cover a significant aspect of each problem discussed in prior sections.

First, we develop efficient performance analysis tools to diagnose systemic idleness in a multi-component system such as CPU-GPU architectures. Second, we develop a technique to alleviate the overhead of barrier synchronization in distributed parallel programs. Third, we develop a novel architecture-aware mutual exclusion algorithm to make locks deliver high throughput on modern many-core shared-memory NUMA machines. Fourth, we develop a fine-grained profiler to pinpoint and quantify losses arising from redundant memory accesses. Finally, we develop an open-source library for efficiently attributing fine-grained execution metrics to source and data in context.

More precisely, in this dissertation, we make the following contributions.

1. We present lightweight, scalable performance analysis techniques for GPU-accelerated heterogeneous architectures. Tools that we developed using these methods have helped identify bottlenecks in important applications as well as helped pinpoint hardware anomalies on leadership-class supercomputers. The techniques developed are planned to be included in the product roadmap of a leading accelerator manufacturer. Prior to this work, "hot spot" analysis was the primary mode of performance analysis on heterogeneous architectures [14, 81, 91, 148, 177, 213]; "idleness" analysis was limited to homogeneous processor environments only [137, 222, 225].

2. We develop a lightweight technique to detect redundant barrier synchronization in Partitioned Global Address Space (PGAS) programs. We demonstrate that redundant synchronization is pervasive in a large modular scientific code base that employs PGAS-style programming model. In addition, we develop an adaptive runtime system based on a sound theory that can automatically elide barriers when possible. Prior to this work, few, if any, efforts were made to elide redundant synchronization [3, 67, 105, 109, 192, 209, 210, 245, 246] in large production parallel programs.

3. We present a novel architecture-aware mutual exclusion algorithm (HMCS lock) that makes better use of locality in multilevel NUMA architectures. The HMCS lock delivers significantly higher lock throughput compared to state-of-the-art locking mechanisms under high contention. We also build precise analytical modeling of various queuing locks and provide proofs for throughput and fairness guarantees of the HMCS over prior designs. An enhancement to the HMCS lock—the AHMCS lock—makes it dynamically adapt to changing contention and deliver top performance under high, moderate, or low contention. Our hierarchical and adaptive algorithm is an addition to the on-going research on NUMA locks [62, 63, 72, 140, 143, 182, 191].

4. We show that wasteful memory accesses are is surprisingly common in complex software systems. We present a novel, efficient, and scalable, fine-grained analysis framework that can pinpoint and quantify wasteful memory operations such as dead writes. Insights from this framework helped improve performance of various code bases by a significant margin. The most relevant prior art in this area was a hardware-based method [34]. Our software-based technique has inspired other researchers to explore inefficiencies caused by redundant computations and such [13, 236].

5. We present a novel approach for efficiently maintaining fine-grained calling context, enabling analysis tools to attribute metrics to both calling context and data. We debunk the myth that fine-grained attribution to calling contexts is infeasible. We developed an open-source call-path collection library for Intel's Pin [144] binary instrumentation framework. The library is an essential component for fine-grained monitoring tools used for performance analysis and software correctness checking. Prior to this

work, attributing context to each executed machine instruction was considered infeasible [69, 229].

Some parts of this dissertation, such as idleness analysis and contextual attribution, develop tools and techniques necessary to understand the performance of programs. Other parts of this dissertation, such as redundant barrier elimination and hierarchical locks, develop solutions that address performance problems.

Performance tools developed in this dissertation focus on resource idleness and wasteful resource consumption. These tools address identifying performance losses arising from load imbalance in parallel programs as well as wasteful memory operations. The adaptive runtimes designed in this dissertation are useful for alleviating parallel overheads incurred due to 1) unnecessary barrier synchronization on PGAS programs and 2) costly mutual exclusion on NUMA machines.

The tools and techniques developed in this dissertation are not only useful but also mandatory for bridging the performance chasm between the capabilities of the modern hardware and the performance achieved by production software systems. Besides performance analysis tools, the tools that help developers write correct parallel programs are of prime importance. Infrastructure developed as part of this dissertation can serve as a foundation for correctness tools [154] necessary for data race detection, memory leak detection, record and replay of threaded programs, and vulnerability detection, among others.

1.4 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 describes assessing the idleness on heterogeneous architectures. Chapter 3 describes the barrier elision technique to alleviate the overhead of redundant barrier synchronization. Chapter 4 describes the HMCS and AHMCS locking protocols to alleviate the overhead of locks for NUMA architectures. Chapter 5 describes a method to assess redundant memory accesses. Chapter 6 describes a framework for fine-grain call-path collection—a recurring theme in this dissertation. Chapter 7 discusses work related to our research. Finally, Chapter 8 summarizes our conclusions along with some avenues for future work.

Chapter 2

Assessing Idle Resources in Hybrid Program Executions

I never remember feeling tired by work, though idleness exhausts me completely.

Sherlock Holmes

The most important aspect of performance tuning a parallel program is ensuring the work is well partitioned to utilize all available resources. Load imbalance and program serialization lead to poor resource utilization. Quantifying resource utilization in the *homogeneous* processor environments has been extensively studied [22, 23, 28, 49, 51, 137, 222, 223, 225, 239]. Quantifying resource utilization in *heterogeneous* processor environments, however, had not been well studied prior to this work. This chapter offers a rich set of tools and techniques for performance analysis of heterogeneous processor environments. We begin this chapter with the motivation for GPU-accelerated architectures and the need for better performance analysis techniques on such architectures. Subsequent sections, gradually develop our performance analysis strategy for GPU-accelerated architectures and demonstrate the effectiveness of our methods.

2.1 Motivation and overview

GPUs have come of age for general-purpose computing. Emerging supercomputers are increasingly employing GPU accelerators [176]. Not only do these GPU-accelerated systems deliver higher performance than their counterparts built with conventional multicore processors alone, but these accelerated systems also deliver improved power efficiency [92]. The increasing use of such GPU-accelerated systems has motivated researchers to develop new

techniques to analyze the performance of these systems.

To develop useful performance tools for GPU-accelerated systems, we needed to answer two questions: 1) what data do we want to collect? and 2) how do we want to collect it?

To date, much of the work on performance analysis of heterogeneous architectures, e.g., [91, 187, 213, 247], has focused on identifying performance problems in GPU kernels. While identifying GPU kernel-level issues is important, this is only one aspect of the larger problem. Whole application performance analysis is equally important for tuning large GPU-accelerated applications. Such analysis requires a system-level view of performance data. Hence, the data collection question reduces to deciding what kinds of *system-level* analyses can best augment standard component-level profiling and tracing [17].

Studies by Luk et al. [145] and Song et al. [217] have demonstrated that dynamically partitioning an application's work between CPU and GPU is important for delivering high efficiency for a variety of applications. Similarly, Aji et al. [6] have demonstrated the significance of application-wide tuning. By understanding the whole-application performance behavior, Aji et al. enhance the overlap between CPU and GPU computations in large MPI applications. These observations inspired us to develop an analysis technique to address the work-partitioning issue. Evaluating the effectiveness of an application's work partitioning is a systemic question. It is not easily addressed by focusing on individual components.

Most performance analysis tools that support GPU-accelerated architectures [81, 148] have employed the classical hot spot analysis technique. As mentioned previously in Chapter 1, the philosophy of hot spot analysis is to inform the user about code regions consuming large amounts of resources such as CPU cycles. Any tool that focuses on *hot spot analysis* can only quantify *where* a program spends its resources. Each component may have different hot spots. In addition, in multi-component systems, identifying where resources are *not* used is equally important for performance analysis. At best, hot spot analysis measures and reports the symptoms of performance problems. Hot spot analysis does not necessarily guide developers toward root causes of performance problems in GPU-accelerated applications. The example shown in Figure 2.1 highlights this point in a heterogeneous environment. If the CPU code executing during the interval labeled `A` cannot be tuned further, then improving `KernelM`, a GPU kernel whose execution is overlapped with `A`, will not shorten the

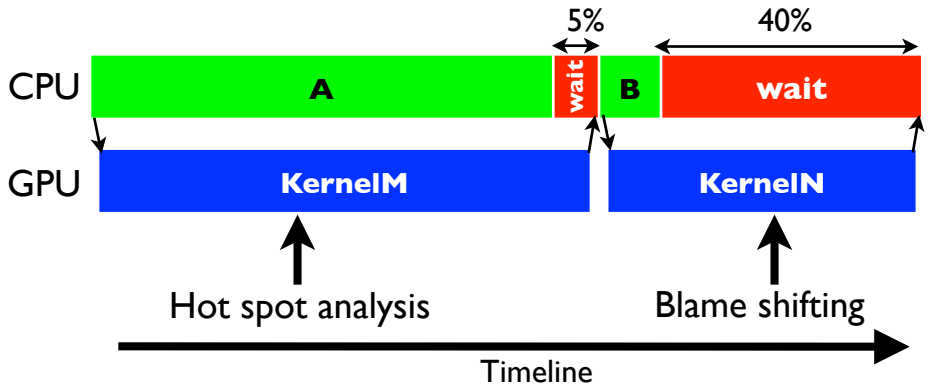


Figure 2.1 : Here we consider a timeline for a hybrid program executing on both a CPU and a GPU. While executing the code fragments A and B, the CPU spends a total of 45% of its time waiting. Tuning the GPU KernelN has the potential for a larger reduction in execution time than tuning the GPU KernelM. Hot spot analysis would identify KernelM, the longer of the two, as the most promising candidate for tuning. Our CPU-GPU blame shifting approach would highlight KernelN since it has a greater potential for reducing CPU idleness.

execution by more than 5%—the time the CPU sits idle awaiting the results of `KernelM`. However, in the same application, the CPU sits idle for 40% of the execution awaiting the completion of GPU `KernelN`; hence, tuning `KernelN` could reduce the execution time by up to 40%. Hot spot analysis would point to `KernelM` as the most time-consuming GPU kernel and thus fails to guide a programmer to `KernelN`. `KernelN` represents a better opportunity for tuning. This problem is exacerbated in full applications with several kernels and more complicated execution schedules.

To address the limitations of hot spot analysis, we supplement it with novel systemic *idleness* analysis. Our idleness analysis identifies *CPU* code regions that cause *GPU* resources to sit idle. Symmetrically, our approach also pinpoints *GPU* kernels that cause *CPU* cores to sit idle. Moreover, our analysis *quantifies* the amount of idleness caused by each offending CPU code region or GPU kernel. Typically, this sort of systemic analysis would require postmortem analysis of execution *traces*. Our analysis, however, requires only a *profile*. The reason one can accomplish idleness analysis without traces is because of a technique that we developed called *CPU-GPU blame shifting*. The key idea behind CPU-GPU blame shifting is to transfer the blame for idleness in one part of the system to concurrent computation elsewhere in the system to analyze how applications use the compute resources of heteroge-

neous architectures. In *CPU-GPU blame shifting*, we instantaneously blame code executing on the non-idle resource (e.g., a kernel executing on a GPU) when a symptom of idleness is detected (e.g., CPU waiting at a synchronization routine).

For the example shown in Figure 2.1, our blame-shifting strategy identifies GPU `KernelN` as a promising target for tuning. Furthermore, our strategy *quantifies* that tuning `KernelN` could improve performance by no more than 40%.

GPU-accelerated programs running on heterogeneous supercomputers face yet another idleness problem. It is a common idiom in GPU programming to block a CPU until some work offloaded to a GPU has finished. However, we noticed that such blocked system calls do not often unblock long after the GPU has completed the work. On large clusters of GPU-accelerated systems, such delays, introduced by stalls in blocked system calls, have deleterious effects on the overall performance of the system. In Single Program Multiple Data (SPMD) programs, where all processes wait at a collective communication, a delay introduced by one process because of a stall in its system call for its GPU work, delays all processes participating in the collective communication. A stall is a special type of idleness; unlike an idle resource, which has no work assigned to it, a stalled process causes a resource to idle while waiting for the completion of an asynchronous task assigned to another resource. Correctly identifying and quantifying this kind of performance problem required us to determine when a given stall was caused by a blocking system call. Unlike the idleness analysis previously described, our stall analysis requires traces. Lightweight traces collected via sampling using HPCTOOLKIT [2] enabled us to implement the stall analysis efficiently.

CPU-GPU blame shifting and stall analysis complement each other and offer different perspectives on idleness analysis. For large-scale applications, while CPU-GPU blame shifting identifies idleness within a node[1] (*intra-node* idleness analysis), our stall analysis identifies idleness across nodes (*inter-node* idleness analysis).

Ideally, we would prefer a low-overhead sampling-based approach for data collection. However, the sampling-based performance measurement on GPUs is either not supported or unsatisfactory. As a result, we had to use instrumentation to allow sampling of GPU performance data. What we developed is a *sampling-driven* approach. In our sampling-

[1]Section 2.6.5 details multiple processes sharing a node.

driven approach, CPU performance data is collected using asynchronous sampling, but the sampling engine takes on the additional responsibility of inspecting GPU state information maintained with instrumentation. We developed this more complex methodology primarily to enable the blame-shifting idleness analysis described previously.

We note that both our analysis techniques and our sampling-driven measurement methodology are fully general. For the work described in this chapter, however, we implemented our ideas to support analysis of CUDA [175] programs.

2.2 Contributions

This work on CPU-GPU blame shifting and the stall analysis, which appeared in the Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13) [43], makes the following contributions.

1. It proposes a sampling-driven CPU and GPU profiling as well as tracing of GPU-accelerated systems,

2. It implements idleness analysis via CPU-GPU blame shifting as well as stall analysis in HPCTOOLKIT,

3. It demonstrates the utility of CPU-GPU blame shifting by identifying opportunities in LULESH-1.0 [110, 111] code and improves its running time by 30%,

4. It demonstrates the utility of our monitoring approach by pinpointing an anomalous GPU in a cluster of 100s of GPUs on the Keeneland supercomputing cluster) [232]—a hybrid cluster with nodes that contain both multicore processors and GPU accelerators,

5. It demonstrates the utility of stall analysis by quantifying up to 18% performance loss because of frequent stalls in large-scale executions of the LAMMPS code, and

6. It highlights the need for better performance analysis support from GPU vendors.

2.3 Chapter roadmap

The remainder of this chapter describes our analysis techniques and their sampling-driven implementation in more detail. The chapter is organized as follows: Section 2.4 gives general background plus a summary of specific work that forms the basis for our tool, Section 2.5 describes our general approach and further details our blame-shifting and stall analysis, Section 2.6 sketches the details of our implementation, Section 2.7 describes case studies that illustrate the effectiveness of our techniques, and Section 2.8 ends with a discussion.

2.4 Background

In this section, we describe diverse areas that constitute the background needed for understanding this work. First, we introduce our terminology. Next, we present challenges for performance measurement tools on heterogeneous systems. Then, we describe both NVIDIA's CUDA [175] programming model and NVIDIA's CUPTI [173] performance tools interface. Finally, we describe the open source HPCTOOLKIT performance tools, which form the basis of our work.

2.4.1 Terminology

We use the term *heterogeneous* to mean architectures that include both CPU and GPU compute resources. We use the term *hybrid* to describe a code that executes in part on CPU(s) and in part on GPU(s). A *task* is an asynchronous GPU activity, such as a kernel or a data transfer. A task is *outstanding* when it has been enqueued for execution but not complete. A task is *active* if it has begun execution but not complete. *Profiling* aggregates performance metrics over time and attributes them to code regions. *Tracing* records execution events in temporal order. For convenience, we refer to our GPU extensions to HPCTOOLKIT as G-HPCTOOLKIT.

2.4.2 Measurement challenges on heterogeneous platforms

Heterogeneous systems today present several challenges for performance tools:

- **Asynchrony:** GPU activities run asynchronously with respect to the CPU and deliver no notification when they start or end.
- **Resource sharing:** A GPU can execute tasks from multiple threads of a process, as well as multiple MPI processes of an application simultaneously.
- **Minimal infrastructure:** Today's GPUs lack hardware support for performance monitoring using asynchronous sampling. Also, current APIs for GPU performance measurement are missing several key capabilities needed to support whole program analysis.

2.4.3 CUDA and CUPTI overview

NVIDIA's CUDA programming model [175] enables programmers to express data parallel computations and map them onto GPUs as tasks. On a heterogeneous architecture, after launching an asynchronous task, a CPU thread can continue execution or immediately wait for completion of the GPU task. CUDA applications manage concurrency through *streams*. A stream is a sequence of tasks that execute in order on a GPU. Tasks from different streams may interleave arbitrarily or run concurrently. CUDA *events* are lightweight markers in a CUDA stream. Events can be queried to inquire about their completion. An event in a stream may not complete until all tasks preceding it in the same stream complete.

NVIDIA's CUPTI performance tools interface [173] is a framework that supports performance analysis of CUDA codes. CUPTI provides the ability to collect GPU hardware counter values. Unlike CPU hardware counters, GPU hardware counters do not deliver an interrupt when they overflow. CUPTI supports attributing hardware counter measurements to GPU activities by providing callbacks upon entry/exit of any CUDA API routine. CUPTI's *Activity API* can trace the execution of tasks. For each task that executes, it logs a record that contains the task start and end times. Activity records can be examined only at synchronization points. To this date, CUPTI has no means to *instantaneously* inform a profiling tool when the following state changes occur:

1. When the GPU is busy executing an application task,

2. When the GPU is idle not working on any task,

3. When an outstanding task becomes active, and

4. When an active task finishes.

2.4.4 HPCToolkit overview

HPCTOOLKIT [2] is an open source suite of tools that supports measurement, analysis, attribution, and presentation of application performance for parallel programs. HPCTOOLKIT has three features that characterize its efficacy for CPU programs. First, HPCTOOLKIT collects almost all performance data using asynchronous sampling of timer and hardware counters, employing instrumentation only when unavoidable. HPCTOOLKIT's overhead for performance data collection is typically less than 5% [224]. Second, HPCTOOLKIT attributes performance metrics to full calling contexts for CPU code. In addition, HPCTOOLKIT maintains these calling contexts in a compact Calling Context Tree (CCT) representation [9], leading to compact profiles and small execution traces [221]. Third, by using on-the-fly binary analysis, HPCTOOLKIT works on multilingual, fully optimized, statically or dynamically linked applications and supports a wide variety of programming models. This on-the-fly binary analysis enables HPCTOOLKIT to attribute costs to both an application and its constituent runtime libraries without instrumenting either of them.

Prior to this work, HPCTOOLKIT provided no support for analysis of hybrid codes. In this chapter, we describe how we address that shortcoming via G-HPCTOOLKIT.

2.5 New analysis techniques

To pinpoint bottlenecks in executions of hybrid programs, we augment HPCTOOLKIT's standard profiles and traces with two new analysis techniques: idleness analysis and stall analysis. Section 2.5.1 covers the idleness analysis. Section 2.5.2 covers analysis of stalls caused by blocking calls to operating system services.

2.5.1 Idleness analysis via blame shifting

In hybrid codes, the execution schedule for CPU and GPU tasks governs tuning opportunities and helps set performance expectations. When code regions are well overlapped, tuning just one part provides little overall performance improvement if the other part is already well tuned. For example, tuning a lengthy GPU kernel is unnecessary if the CPU only requires its results long after completion of that kernel. Tuning code regions whose executions do not overlap, however, can reduce the critical path and thereby improve the overall running time. In hybrid codes, when a GPU is idle (symptom), CPU threads (cause) are responsible for not keeping it busy with a sufficient number of asynchronous tasks. Similarly, when a CPU thread is waiting (symptom) for GPU tasks to produce results, the active GPU tasks (cause) are responsible for blocking the CPU thread from making progress.

Our approach for identifying causes of idleness in heterogeneous applications was inspired by the *blame shifting* technique pioneered by Tallent et al. [225]. Tallent et al. were concerned with lock contention. Their key insight was to attribute blame to lock *holders* for the idleness of threads waiting for lock acquisition. We apply this idea of shifting blame for idleness in one part of the system to concurrent computation elsewhere in the system to analyze how applications use the compute resources of heterogeneous architectures. The result is a strategy we call *CPU-GPU blame shifting*. In *CPU-GPU blame shifting*, we blame code executing on the non-idle resource (e.g., a kernel executing on a GPU) when a symptom of idleness is detected (e.g., CPU waiting for kernels to complete on an NVIDIA GPU using `cudaDeviceSynchronize`). We *quantify* the causes of idleness as follows:

- CPU code regions that execute when a GPU is idle accumulate blame proportional to the time the GPU was idle during their execution.
- GPU tasks that execute when a CPU thread is awaiting their completion accumulate blame proportional to the time the CPU thread waited for their completion.

In practice, the wall clock time or cycles for which a resource is idle represents the source of "blame". Code region(s) that are frequently responsible for causing idleness accrue more blame than other regions and thereby warrant the programmer's attention. We characterize hybrid codes with the metrics presented in Table 2.1.

Metric name	Qualification (when)	Context of attribution	Quantification (amount)
CPU_IDLE (CI)	CPU is waiting for GPU	CPU code region e.g. cudaDeviceSynchronize	Time spent waiting for GPU
CPU_IDLE_CAUSE (CIC)	CPU is waiting for GPU	Launch contexts of active GPU tasks (blame)	Time the task was active when CPU was waiting
GPU_IDLE_CAUSE (GIC)	GPU is idle	CPU code regions (blame)	Time when GPU was idle
GPU_EXECUTION_TIME (GET)	Length of GPU task	Launch context of the GPU task	Time taken for the GPU task
H_TO_D_BYTES (H2D)	Host to device data xfer	Launch context of the CPU-to-GPU data transfer	bytes transferred
D_TO_H_BYTES (D2H)	Device to host data xfer	Launch context of the GPU-to-CPU data transfer	bytes transferred

Table 2.1 : Metrics for performance analysis of hybrid codes. The **Qualification** column shows **when** the attribution happens to a particular metric. The **Context of attribution** column shows **which** code context accumulates the metric. The **Quantification** column shows **how much** quantity is accumulated in the metric at a particular context. GPU_IDLE_CAUSE and CPU_IDLE_CAUSE are the only two metrics that are computed using blame shifting.

CPU_IDLE_CAUSE (CIC) and GPU_IDLE_CAUSE (GIC) are the blame shifting (cause) metrics. CPU_IDLE (CI) is an idleness symptom metric. GPU_EXECUTION_TIME (GET), H_TO_D_BYTES (H2D), and D_TO_H_BYTES (D2H) are useful hot spot metrics. Each metric adds up to provide the total time spent in that state. We attribute each metric to a full call-path; hence the programmer can easily identify inclusive and exclusive costs at each level in the call-path. A key attraction of this technique is that the analysis and attribution can be performed on the fly without having to record full execution traces.

Blame shifting precisely identifies the code that lies on the critical path in a two-component system. In multi-component systems, however, blame shifting identifies a superset of code on the critical path.

In this work, we focus only on CPU-GPU idleness. We do not consider the cases when a CPU-thread is idle waiting for another CPU-thread or I/O activity. Also, in this dissertation we do not explore gathering GPU hardware performance counter measurements of individual kernels. HPCTOOLKIT uses the CUPTI interface to gather GPU performance counter values for individual kernels, which is not the key focus of our work.

2.5.2 Stall analysis

Our stall analysis technique detects when performance is degraded by undue delays in blocking system calls. Even if excessive blocking infrequently occurs in each process, it can

dramatically degrade the performance in parallel systems [188]. For example, suppose one process is delayed because of a blocking system call. If the delay occurs before arrival at a barrier, then the entire application must wait for the late arriver. We emphasize that we are seeking *sparse* blocking phenomena. If *most* processes are delayed before arriving at a barrier, this set of blocking events is not a candidate for stall analysis. Non-sparse blocking phenomena are easily captured as part of blame shifting. Moreover, the sparse stall phenomena may repeat many times while an application executes. Our stall analysis calculates the cumulative effect of such stalls. Unlike blame shifting, stall analysis is performed via postmortem inspection of execution traces.

Detecting a sparse stall event consists of three parts listed below.

1. Detect that a stall has occurred in at least one process,
2. Confirm that the set of stalls for the given time interval is sparse, and
3. Confirm that the sparse set affects the performance of the full application.

To detect that an undue delay in a blocking system call has occurred in a given trace, we rely on a regularity property of time-based periodic sampling—samples should occur at regular intervals. While small variations in the length of sampling intervals is normal, any "significant" lengthening of the time intervals between samples indicates that the periodic timer interrupts are not being reported by the OS, which means that blocking has occurred in that interval.

To determine when a given interval in a trace is unusually long, we find the median interval span in the given trace. We mark a span as unduly long if its length is more than 10 times the median. This technique is a heuristic that worked for us. Other heuristics, such as Manhattan distance used by Google-wide profiling [196], would also yield useful results.

To determine if a given unduly long interval is part of a sparse set, we count the total number of unusually long intervals in each trace that occur within the same time interval. If less than 10% of the traces have unusually long intervals for the time period in question, then we consider this set of unusually long intervals to be a sparse set.

Finally, to determine if a sparse set affects overall system performance, we check the calling context of each sample in the set of traces that are *not* in the sparse set. If each

sample has a barrier or collective operation in the calling context, then the sparse set meets all the criteria to be deemed an unusually long stall. We acknowledge that the 10x median rule for "unusually long" and the less than 10% rule for classifying a set as sparse are ad hoc. These should be tunable parameters. In our prototype, however, we restricted ourselves to the aforementioned thresholds.

To quantitatively assess the cumulative impact of all unusually long stalls, we merge delays from two or more processes that have overlapping stalls and consider it as one larger stall starting at the earliest start time and ending at the latest end time of the overlapping stalls. It is easy to accomplish this by using interval trees to compute the union of the overlapping time ranges. The aggregate value of non-overlapping time ranges accumulated in the interval tree provides the total system-affecting blocking time and its ratio with respect to the total execution time of the program provides the percentage of time the execution was impacted due to blocking system calls.

2.6 Implementation

We organize the implementation details of our idleness analysis as follows: Section 2.6.1 describes a basic mechanism for our implementation of CPU-GPU blame shifting. Section 2.6.2 details a strategy to accommodate blame shifting inside a CUDA call. Section 2.6.3–2.6.5 gradually develop CPU-GPU blame shifting for multi-stream, multi-thread, and multi-process shared GPU configurations. Finally, Section 2.6.7 describes support for GPU tracing in HPCTOOLKIT. In the rest of this chapter, we use the terms "CPU-GPU blame shifting" and "blame shifting" interchangeably.

2.6.1 Basic CPU-GPU blame shifting

G-HPCTOOLKIT employs sampling-based performance measurement of code running on each CPU core in combination with lightweight instrumentation of GPU operations implemented using NVIDIA's CUDA programming model [175]. In this subsection, we sketch the details of sampling-driven blame shifting in the simplest setup, where the execution involves only a single-threaded CPU process along with one GPU steam. Figure 2.2 pro-

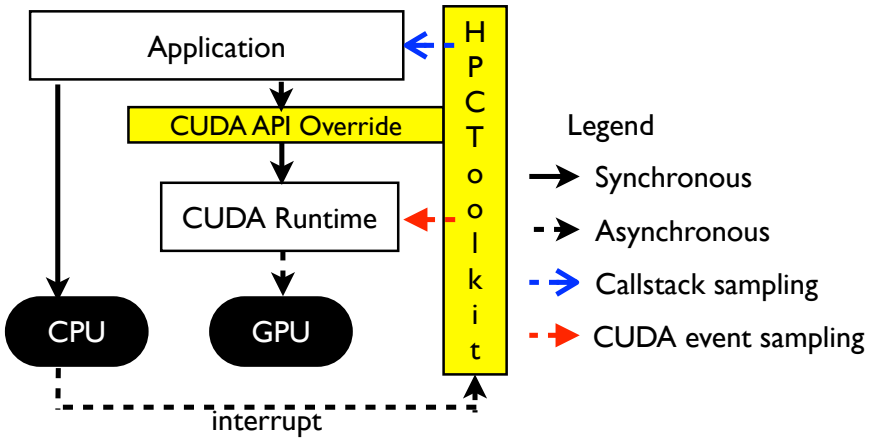


Figure 2.2 : Schematic diagram of G-HPCTOOLKIT with asynchronous sampling of CPU and instrumentation of CUDA APIs to support sampling of GPU activities. A CPU interrupt (sample) causes the HPCTOOLKIT to asynchronously query the status of outstanding GPU tasks. A few GPU APIs called from a CPU thread also invoke the HPCTOOLKIT to synchronously query the status of outstanding GPU tasks.

vides a schematic view of G-HPCTOOLKIT. To monitor asynchronous tasks on NVIDIA GPUs, we use a two-part technique. First, we insert CUDA *events* before ("start") and after ("end") all GPU task launches. This technique is similar to the "Event Queue Method" independently developed by Malony et al. [148]. Inserting these events requires instrumenting CUDA task-launch functions. While CUPTI provides suitable instrumentation hooks, our implementation wraps CUDA's kernel launch API using library interposition [164]. In addition, we wrap functions that manage streams, allocate memory, move data, or perform synchronization. We chose this method to circumvent host-thread serialization that occurs when using pre-5.0 versions of CUPTI.

The second part of our two-part technique involves querying the status of these events. Querying takes place periodically when a CPU sample occurs. At any time, we infer that a task is active if its "start" event is complete but its "end" event is incomplete. A task is complete if its "end" event is complete. Querying can happen either during sampling or inside one of the CUDA-functions we wrap. The time elapsed between the completion of the "start" and "end" events surrounding a task, provides a close estimate of the execution time of that task. It is worth mentioning that events, in addition to a complete/incomplete

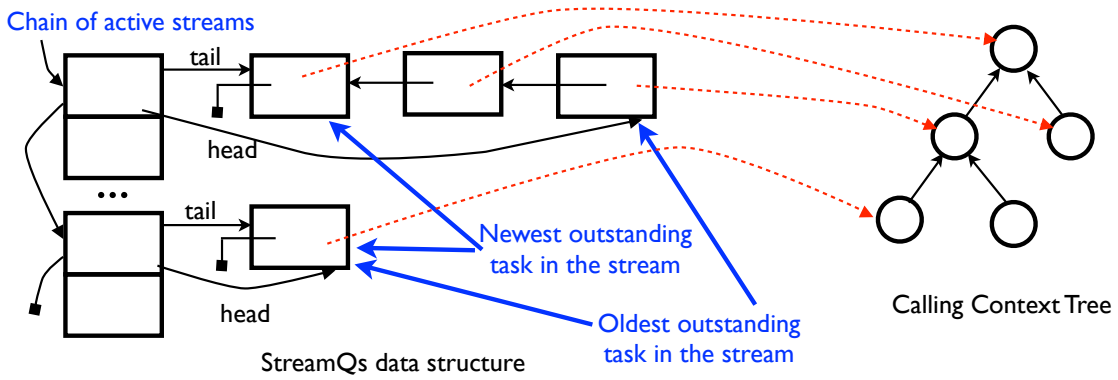


Figure 2.3 : Data structure used to maintain the association between GPU tasks and CPU contexts.

boolean flag, carry a completion timestamp, which can be queried any time after event completion. To confirm the accuracy of our technique, we compared the deviation in kernel timings as measured by CUDA events vs. CUPTI's Activity API, for 26 unique kernels in the LULESH 1.0 [110] benchmark. At the sample rate of one per millisecond, the geometric mean of percentage deviations from the precise data collected via CUPTI's ActivityAPI [173] over all the LULESH kernels was ∼1%, which we deem acceptable.

We store the "start" and "end" events of each task, along with a pointer to the call-path that launched the task, in an auxiliary data structure that we call `StreamQs`. G-HPCToolkit's `StreamQs` data structure, shown in Figure 2.3, is an array of queues, where each queue represents the tasks issued on a GPU stream. The head and tail of each queue in `StreamQs` represent the oldest and newest outstanding tasks issued on that stream respectively. A new GPU task can be enqueued at the tail of a queue in constant time. On each sampling event that happens on a CPU thread, the CPU queries the completion status of the "end" event of the task at the head of each active stream, which can be accessed in constant time. If the "end" event is complete, the query moves forward through the list until it reaches an incomplete event in that stream. `StreamQs` may be accessed and updated *synchronously* or *asynchronously* by all threads in a process. *Synchronous* accesses happen during the execution of instrumentation inserted inside wrapped functions, whereas *asynchronous* accesses happen during sampling interrupts.

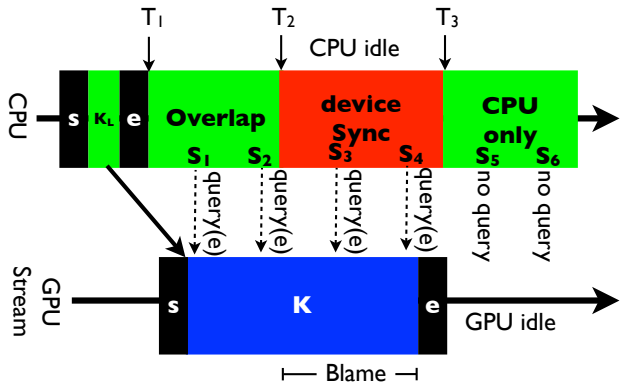


Figure 2.4 : Proxy-based GPU sampling. Current generation GPUs do not deliver samples, unlike CPUs. To sample GPU activities, G-HPCTOOLKIT relies in the periodic samples taken by CPU threads. On each CPU sample, G-HPCTOOLKIT queries the GPU to infer the status of outstanding tasks.

Figure 2.4 sketches our approach for sampling-based performance analysis of GPUs, for a single CPU-thread and a single GPU-stream case. We use the `StreamQs` data structure to help transfer blame to the appropriate CPU and GPU contexts. Let the calling context at a point P in CPU execution be represented as $c(P)$. Let the calling context for the CPU-side launch point of a GPU task K be represented as $c(K_L)$. At $c(K_L)$, in Figure 2.4, G-HPCTOOLKIT inserts the start event s and end event e surrounding the task K through the wrapped `cudaLaunch` API. A new node $< s, c(K_L), e >$ is enqueued into a queue representing the appropriate stream in the `StreamQs` data structure.

At time T_1, the CPU starts to execute the code marked as *overlap* concurrently with kernel K on a GPU stream. On interrupts S_1 and S_2 during this interval, G-HPCTOOLKIT queries the status of the end event e for the oldest outstanding task K and infers that K is active in this interval. Therefore, no blaming occurs during this stage.

At time T_2, CPU calls `cudaDeviceSynchronize` to wait for the completion of tasks. G-HPCTOOLKIT's wrapped version of the function sets a thread-local flag (`isAtSync`) indicating that the CPU thread is idle. The blame attribution in this region follows the deferred blame-shifting strategy discussed in Section 2.6.2, which leads to blaming kernel K with the amount $(T_3 - T_2)$ as the cause of CPU wait time. This attribution increments the `CPU_IDLE_CAUSE` metric for $c(K_L)$ by $(T_3 - T_2)$.

At T_3, the return from `cudaDeviceSynchronize` causes G-HPCTOOLKIT to query e and infer the completion of task K, thereby dequeueing it from `StreamQs`. Also, we unset the `isAtSync` flag indicating that the CPU is active. Sampling at S_5 notices no outstanding GPU-tasks and declares the GPU idle. Samples taken at S_5 and S_6, blame their respective CPU contexts for keeping the GPU idle, which is accomplished by incrementing their `GPU_IDLE_CAUSE` metric.

2.6.2 Deferred blame shifting

To sample GPU events, G-HPCTOOLKIT needs to call `cudaEventQuery` from within a signal handler. When an asynchronous sample occurs, if the CPU thread is inside a CUDA API function such as `cudaDeviceSynchronize` that already holds a CUDA runtime lock, then calling `cudaEventQuery` will cause it to attempt to acquire the same lock leading to a deadlock. Nvidia foresees no remedy for the deadlock to support GPU sampling.

To avoid deadlock in the aforementioned circumstance, we devised a technique that we call *deferred blame shifting*. In deferred blame shifting, when an asynchronous sample occurs, if the thread is already inside a CUDA call, which can be inferred by inspecting `isAtSync` flag, G-HPCTOOLKIT's timer interrupt handlers do not query whether any outstanding tasks are complete and hence the profiler remains in a blind spot with respect to GPU activities. In the example in Figure 2.4, the time interval between T_2 to T_3 is a blind spot. On returning from a CUDA API call, the wrapped function examines outstanding GPU tasks in each active stream of the `StreamQs` data structure looking for ones that are complete and proportionately blames them for causing the CPU idleness. The ability of events to record completion timestamps for a later query comes in handy here.

To illustrate the deferred blame shifting mechanism, consider Figure 2.5. From time $Sync_s$ to $Sync_e$, our tool cannot inquire about GPU activity. Just before returning from the wrapper for `cudaDeviceSynchronize`, however, we query for the completion of $K1$ and $K2$. We increment the `CPU_IDLE` metric by $(Sync_e - Sync_s)$, attributing the idleness to the CPU context that called `cudaDeviceSynchronize`. Having obtained the start and end times of $K1$ and $K2$, we now *blame* CPU idleness (by incrementing `CPU_IDLE_CAUSE` metric) at $c(K1_L)$ for the amount $(T_2 - Sync_s)$ and $c(K2_L)$ for the amount $(T_3 - T_2)$. The blame for

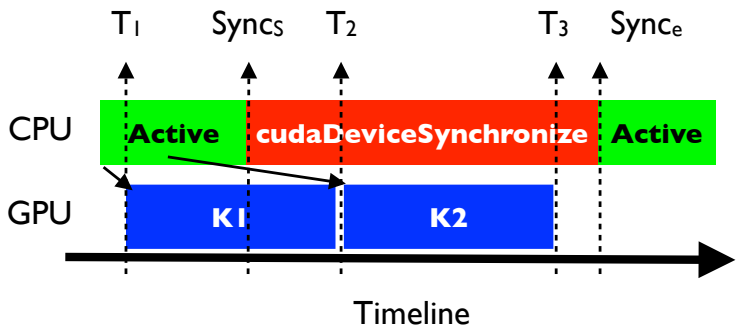


Figure 2.5 : Deferred blame attribution: A technique to attribute blame to GPU kernels when CPU is inside a blind spot such as `cudaDeviceSynchronize`.

the $(Sync_e - T_3)$ interval is still unattributed—both CPU and GPU are idle here. We note that this part of the idleness is the delay between the end of GPU activity and the return from the CUDA synchronization call waiting on it. In our blame-shifting methodology, we consider this delay to be primarily GPU idleness, and we blame it on the CPU context calling `cudaDeviceSynchronize`. A later case study in Section 2.7.1 with LULESH demonstrates a scenario where such a situation arises with `cudaFree`.

2.6.3 Blame shifting with multiple GPU streams

When concurrent kernels are active on multiple streams, CPU idleness is apportioned across all active kernels. Consider the situation shown in Figure 2.6, where the CPU thread is idle from time $Sync_s$ to $Sync_e$. Three kernels `K1`, `K2`, and `K3` overlap with this region. Here, the CPU launches `K1` on stream 2, `K2` on stream 1, and `K3` on stream 2, in that order before going into a wait via `cudaDeviceSynchronize`. Blame attribution for each kernel is as follows:

- Blame on `K1` $= (\alpha) + (\frac{\beta}{2})$. α represents the part of `K1` solely responsible for keeping the CPU idle; β represents the part of `K1` overlapped with `K2`, and hence both kernels share the blame.

- Blame on `K2` $= (\frac{\beta}{2}) + (\gamma) + (\frac{\delta}{2})$. The first term represents the part of `K1` overlapped with `K2`, and hence the blame is shared. The second term represents the part of `K2` solely responsible for keeping the CPU idle. The third term represents the part of `K2` overlapped with `K3`, and hence the blame is shared again.

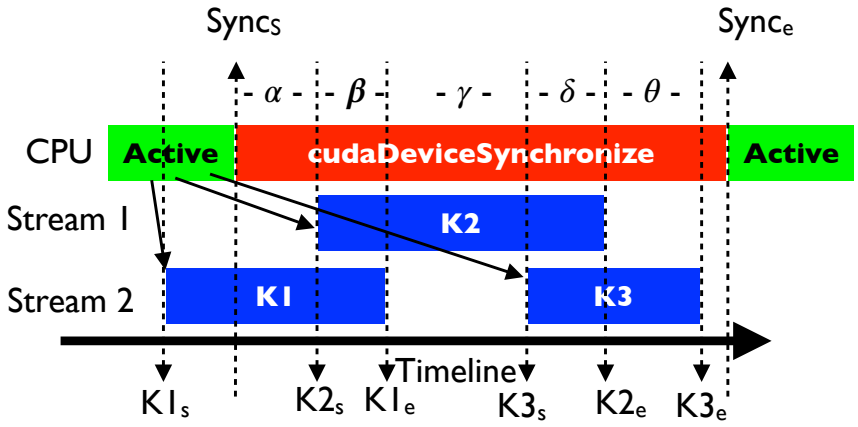


Figure 2.6 : Blame attribution with multiple streams.

- Blame on `K3` $= (\frac{\delta}{2}) + (\theta)$. The first term represents the part of `K3` overlapped with `K2`, and hence the blame is shared. The second term represents the part of `K3` solely responsible for keeping the CPU idle.

As before, the delay in return from synchronization ($Sync_e - K3_e$) is regarded as GPU idleness and blamed on the CPU.

The technique can be implemented in $O(n \log n)$ time complexity by sorting the start and end times of all tasks overlapped in a blind spot, where n is the number of active tasks in the blind spot.

Different CUDA synchronization APIs have different properties. For example, `cudaDeviceSynchronize` waits for all streams to finish; when this API is invoked, we blame CPU idleness on kernels on all streams. On the other hand, `cudaStreamSynchronize` waits for a particular stream to finish, in which case we blame tasks active only within the stream being waited.

2.6.4 Blame shifting with multiple CPU threads

In multi-threaded processes, two cases need to be handled. First, each thread independently blames its CPU context when it observes GPU inactivity. Second, if multiple threads are waiting for one or more GPU kernels to finish, each thread independently blames each kernel that caused its idleness by the amount proportional to that thread's wait time. Multi-thread

and multi-stream are composable. The ability to share `StreamQs` helps us achieve blame shifting easily in multi-threaded applications.

2.6.5 Blame attribution for shared GPUs

Software on supercomputers supports running multiple MPI ranks per node sharing a single GPU. Processes do not have a global view of GPU utilization when the same GPU device is shared by multiple MPI ranks, because of different address spaces. In such scenarios, a GPU is idle if and only if none of the processes sharing the same GPU have any outstanding kernels scheduled on the GPU. To correctly identify GPU idleness, we introduce shared counters (`numOutstandingTasks`), one per physical GPU on a node. The shared counters are allocated in a shared memory segment created using the `shmem` capability in Linux. Each time a task is issued to a GPU, the `numOutstandingTasks` counter associated with that physical GPU is atomically incremented; the counter is atomically decremented when the task finishes. If the `numOutstandingTasks` is non-zero, then the GPU is busy on behalf of some process. G-HPCTOOLKIT will not blame the CPU code on any of the processes sharing the same physical GPU if the associated `numOutstandingTasks` counter is non-zero. Similarly, to blame CPU idleness on a GPU task issued by other processes sharing the same physical GPU, we employ another shared counter (`numIdleThreads`). The `numIdleThreads` counter is atomically incremented/decremented on entry/exit to/from an idle region (e.g.,`cudaDeviceSynchronize`) by each CPU thread. During execution of a task K, its launching process inspects the `numIdleThreads` counter associated with the physical GPU on which K is running and proportionally blames K for keeping other processes (sharing the same physical GPU) idle.

2.6.6 A limitation of current implementation

A known limitation of using events for time measurement is the following: consider two kernels, `K1` and `K2`, along with their start and end events `<s1, K1, e1>` and `<s2, K2, e2>` on two different streams. Let the kernels execute serially, and let the interleaving order be `s1`, `s2`, `K1`, `e1`, `K2`, and `e2`. In such cases, G-HPCTOOLKIT can be fooled into assuming that `K1` and `K2` executed concurrently.

2.6.7 Lightweight traces for hybrid programs

Profiling collects aggregate metrics of execution. Profiling does not capture the time-varying behavior of an execution. It is important to capture the temporal behavior of a program to identify bottlenecks such as serialization.

Tracing involves recording events with timestamps to analyze how an execution unfolds over time. G-HPCTOOLKIT traces CPU-side execution by leveraging the existing infrastructure in HPCTOOLKIT [221]. To monitor CPU activity, HPCTOOLKIT logs a trace record each time a thread receives an asynchronous sample. HPCTOOLKIT's traces consist of a sequence of records, where each record contains a timestamp and the index of a node in a CCT. The path from that node to the root of the CCT represents a full calling context. Each call-path is stored only once in the CCT. Because of compact trace representation, the size of HPCTOOLKIT's trace for a thread is proportional to the number of samples it receives during execution.

In contrast to the sampling-based traces HPCTOOLKIT logs for CPU activity, we trace GPU activities by logging records (time and calling context pairs) at the beginning and end of GPU tasks on each CUDA stream. At present, all tasks on a stream are logged into a trace buffer when we notice their completion. Using this approach, the volume of GPU traces for an application is proportional to the number of GPU tasks the application executes. When GPU tasks complete at a rate higher than a chosen sampling rate, we could reduce trace sizes by dropping information about some GPU activities and logging tasks only at a rate proportional to the sampling frequency. A technique we envision is to record only the most recently finished activity on each stream in `StreamQs` into our traces and drop others that might have finished between the last sample and the current sample. This technique would result in sampled traces analogous to those HPCTOOLKIT records for CPU activity. It is worth noting that blame shifting is agnostic to trace collection and hence it is unaffected if we drop trace records.

Compared to other tracing tools, our traces are both rich and lightweight. They are rich in the sense that each trace record represents a full calling context. Our trace records enable us to analyze and visualize an execution at multiple levels of abstraction, i.e., different call-

path depths. Our traces are lightweight in the sense that each trace record itself is compact (only 12 bytes). Our approach of using the index of an out-of-band CCT node to represent the calling context for a trace record enables us to avoid tracing procedure entry and exit events to recover calling context. For GPU-accelerated program executions, HPCTOOLKIT's `hpctraceviewer` graphical user interface presents a trace line for each CPU thread and each GPU stream. For an MPI program, `hpctraceviewer` presents such a set of trace lines for each MPI rank.

2.7 Evaluation

In this section, we apply G-HPCTOOLKIT to several examples to evaluate its efficacy and runtime overhead. To test the efficacy of our techniques, we followed an uniform analysis methodology. For each test case, we profiled it using G-HPCTOOLKIT with CPU-GPU blame shifting activated. Also, we collected traces each time. If the insights gained from these techniques pointed to an opportunity for improvement, we implemented it. If no solution was apparent but the evidence suggested delays related to blocking system calls, then we ran our stall analysis to confirm (and quantify) the effect of these delays.

In Sections 2.7.1–2.7.3, we present case studies that demonstrate unique insights that G-HPCTOOLKIT provides. In Section 2.7.4, we give a *preliminary* (essentially anecdotal) evaluation of the monitoring-time overhead of G-HPCTOOLKIT. We performed most of our experiments on Georgia Tech's Keeneland Initial Delivery System (KIDS) [232], which is a 120-node HP SL-390 cluster with a Qlogic QDR InfiniBand interconnect. Each node has two Intel Xeon X5660 hex-core CPUs, 24GB memory, and three NVIDIA M2090 GPUs. We also performed some experiments on ORNL's Titan—a Cray XK7 supercomputer. Titan has 18,688 compute nodes linked by Cray's Gemini interconnect. Each compute node has a 16-core AMD Interlagos processor and an NVIDIA Tesla K20X GPU.

2.7.1 Case study: LULESH

As part of DARPA's UHPC program, Lawrence Livermore National Laboratory developed the Livermore Unstructured Lagrange Explicit Shock Hydro dynamics (LULESH) mini-

application [110, 111]. LULESH is an Arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem for one material in 3D. In this section, we study the CUDA-accelerated version of LULESH 1.0. Analysis of executions of CUDA-accelerated version of LULESH 1.0 with G-HPCTOOLKIT provided us with the following insights:

- The CPU is idle 62% of the wall clock time, and 86% of that time is spent inside `cudaFree`,
- The GPU is idle 35% of the wall clock time. `cudaFree` and `cudaMalloc` called from various contexts account for 48% and 47% (a total of 95%) of GPU idleness respectively, and
- There is negligible overlap between the CPU and GPU.

Examining the code regions *causing* CPU idleness (not shown) and GPU idleness (shown in Figure 2.7) using HPCTOOLKIT's `hpcviewer` provides a clue that the developer used `cudaFree` as a synchronization construct analogous to `cudaDeviceSynchronize` in addition to using it to free the allocated device memory. The pattern of repeatedly allocating and freeing device memory was pervasive in the code and was executed several times in each time step. When sorted by the `GPU_IDLE_CAUSE` metric, `hpcviewer` pinpointed the call sites that repeatedly invoked `cudaFree` and `cudaMalloc` as the root causes of GPU idleness.[2] NVIDIA engineers confirmed that `cudaFree` has undocumented dual functionality i.e., it performs `cudaDeviceSynchronize` waiting for all tasks on the GPU to finish, followed by freeing the memory [58]. The CPU-side wait during the synchronization point for the completion of kernels is the primary cause of CPU idleness, *however after synchronization, the CPU is busy freeing the device memory during which the GPU is idle.* Similarly, repeated `cudaMalloc`s are also a cause of GPU idleness. Small amounts of GPU idleness scattered across tens of `cudaFree/cudaMalloc` calls in each iteration add up to contribute ∼30% GPU resource idleness. The blame shifting quantifies the expected speedup by eliminating this idleness to be ∼30%.

[2]In the figure, we show only the top most contributors, but there are other places where the pattern repeats.

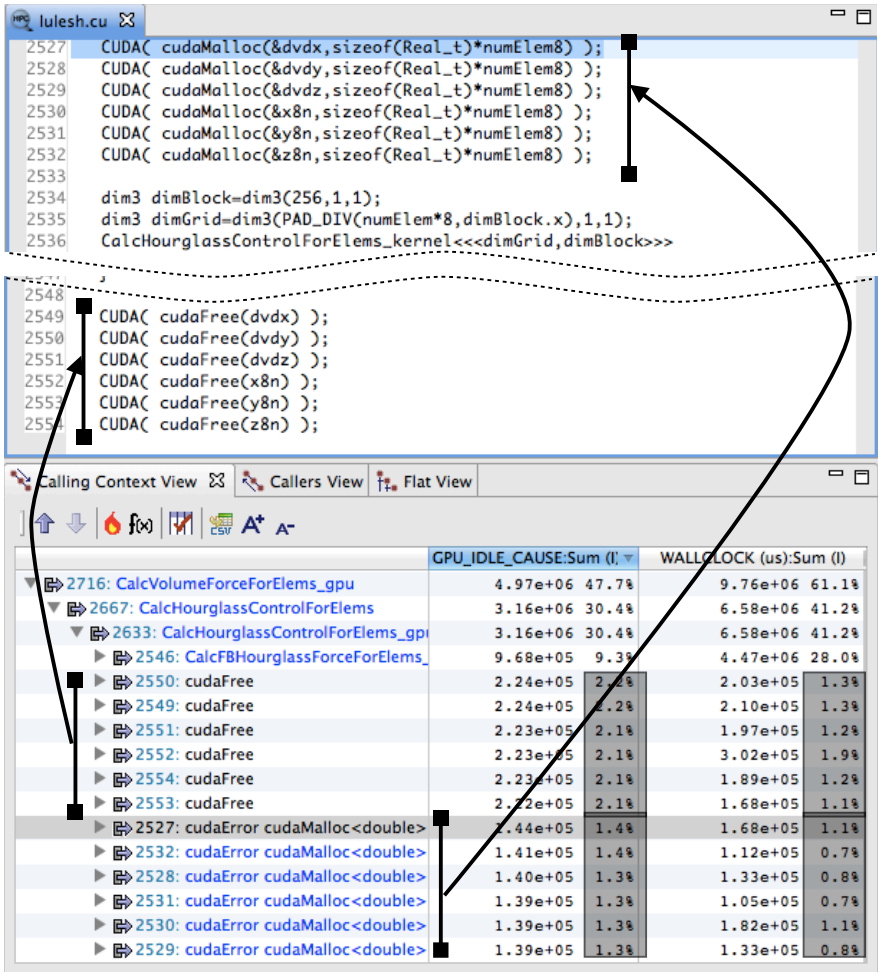


Figure 2.7 : Code-centric view of LULESH with `hpcviewer`. The blame shifting metric `GPU_IDLE_CAUSE` pinpoints `cudaMalloc` and `cudaFree` called from various locations in the program as responsible for vast amounts of GPU idleness.

Figure 2.8 shows LULESH's pattern for allocating device memory as a function of time. In each time step, at each allocation site, exactly the same amount of device memory is allocated and freed. To eliminate GPU idleness caused by repeated device memory management calls, we hoisted them all outside of the main time step loop. There, we allocated a sufficient number of blocks of sufficient size to meet the peak memory demand. Finally, to replace the synchronization that `cudaFree` provided, we explicitly called `cudaStreamSynchronize`. *This optimization—hoisting, pre-allocation, and explicit synchronization—reduced the running time for LULESH by 30%.* We reiterate that G-HPCToolkit's `GPU_IDLE_CAUSE` metric identified calls to `cudaFree` *in the application code.* Besides the running-time benefit,

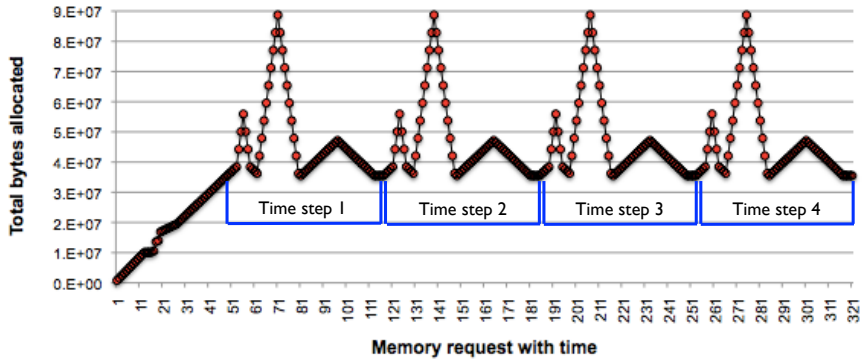


Figure 2.8 : LULESH device-memory allocation graph. Exactly the same amount of GPU memory allocated and freed in a fix pattern in each time step.

Order by `CPU_IDLE_CAUSE`	Kernel	`CPU_IDLE_TIME` μ sec	`GPU_EXEC_TIME` μ sec (% of total)	Rank if sorted by `GPU_EXEC_TIME`
1	`CalcFBHourglassForceForElems`	2.31e+06	2.31e+06 (23.9%)	1
2	`CalcHourglassControlForElems`	8.49e+05	1.10e+06 (11.4%)	4
3	`IntegrateStressForElems`	6.73e+05	6.76e+05 (7.0%)	5
4	`AddNodeForcesFromElems2`	4.16e+05	4.16e+05 (4.3%)	6
5	`AddNodeForcesFromElems`	3.88e+05	3.88e+05 (4.0%)	7
6	`CalcKinematicsForElems`	3.17e+05	1.74e+06 (18.0%)	2

Table 2.2 : Blame shifting vs. hot spot analysis for LULESH kernels.

the optimizations improved LULESH's GPU utilization from 65% to 95%. LLNL researchers had independently identified the same performance bottlenecks and admitted a large manual effort in doing so, whereas with *CPU-GPU blame shifting*, we systematically and automatically identified performance issues in a fraction of time and effort for an unfamiliar code.

Table 2.2 shows the top six kernels of LULESH ordered by their cause for CPU idleness (`CIC` metric). The table also shows their GPU execution time (`GET` metric) and what would have been their rank order (last column) had they been ordered by `GET`—a hot spot analysis metric. It is worth observing that the two orderings are quite different. In particular, note that the 2^{nd} rank kernel (`CalcHourglassControlForElems`) in the blame-shift ordering appears as the 4^{th} rank in the hot spot ordering. This ranking discrepancy tells us that the `CalcHourglassControlForElems` kernel (ranked 2^{nd} by blame shifting) is a more promising candidate for tuning than the `CalcKinematicsForElems` kernel (ranked 2^{nd} by hot spot analysis).

2.7.2 Case study: LAMMPS

LAMMPS [189] is a molecular dynamics code developed by Sandia National Laboratories that simulates biomolecules, polymers, materials, and mesoscale systems. The principal simulation computations in LAMMPS are *neighbor calculation*, *force calculation*, and *time integration*. LAMMPS is parallelized using spatial-decomposition techniques. LAMMPS employs MPI as the distributed computation infrastructure. LAMMPS-GPU [29] is an accelerated version that offloads neighbor and force computations to GPUs while performing time integration on CPUs.

In Section 2.7.2.1 we describe how G-HPCTOOLKIT helped us identify a GPU hardware problems on KIDS. Also, in Section 2.7.2.2 we demonstrate G-HPCTOOLKIT's capabilities for pinpointing scalability losses.

2.7.2.1 Pinpointing hardware performance problems

Figure 2.9 shows G-HPCTOOLKIT traces from a 64 MPI process execution of LAMMPS-GPU for the Lennard Jones (LJ) benchmark. Each GPU is shared by two processes. Each process has two GPU streams. We filtered the results to show only one GPU stream of each process. A high-level view of the traces shows anomalous behavior on GPU streams associated with two of 64 MPI ranks. Zooming in on these two processes (MPI processes 24 and 25), shows that their host-device data transfers were significantly slower than those of their peers (MPI processes 23 and 26). However, their kernel executions were comparable. Metrics for data copies between host and device (`H2D` and `D2H`) for these MPI ranks indicated no difference in the data transfer volume compared to other ranks. This investigation indicated that there might be a hardware problem with the PCI-e bus linking the CPU and GPU on that node hosting those ranks. The KIDS system administrators confirmed that our job was scheduled on the node `kid058` that was delivering low PCI-e bandwidth because of an improperly seated GPU. These two slow processes were in turn slowing down the entire execution since other processes needed to wait for their results at the end of each time step. G-HPCTOOLKIT's lightweight traces in conjunction with the data copy (`H2D` and `D2H`) metrics highlighted the anomalous behavior of the malfunctioning hardware.

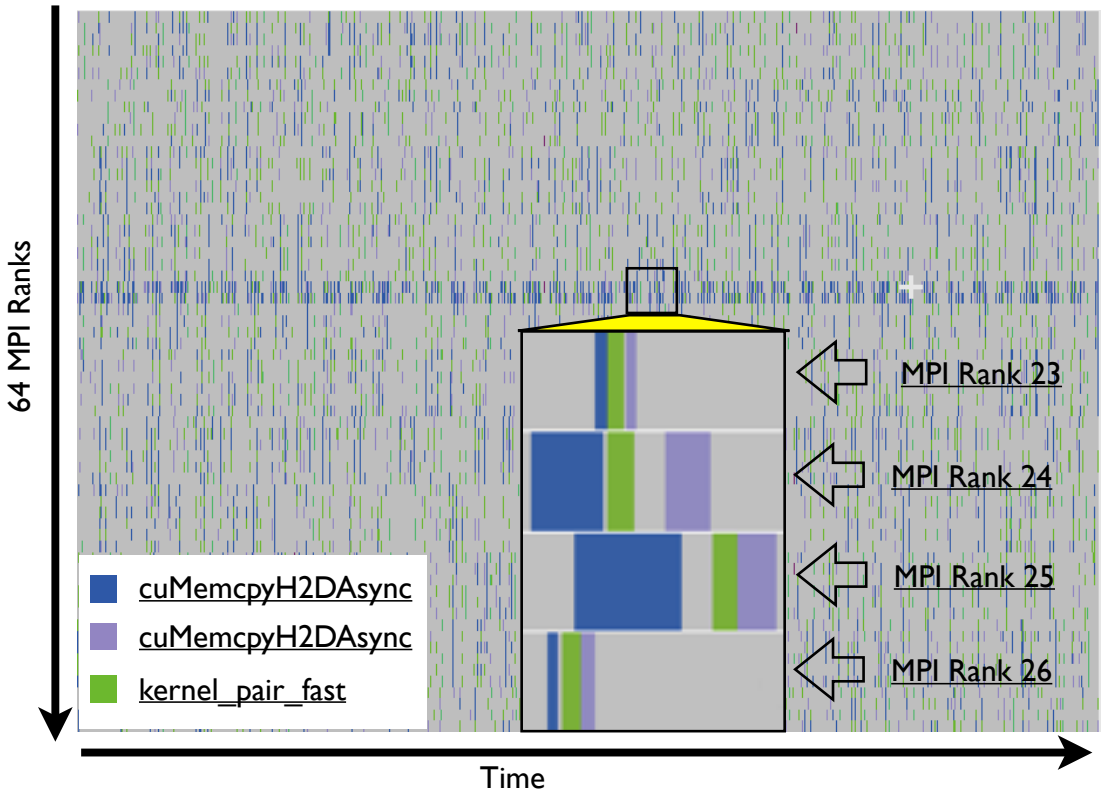


Figure 2.9 : Slow GPUs delaying LAMMPS on KIDS. MPI ranks 24 and 25 are assigned to a GPU that is delivering low PCI-e bandwidth. Hence, the data transfers on ranks 24 and 25 are significantly slower compared to their peer MPI ranks 23 and 24. Kernel executions, however, are not very slow on the affected GPU.

2.7.2.2 Pinpointing scalability limiting factors

Figure 2.10 represents the traces from a 64 MPI process execution of LAMMPS-GPU for the LJ benchmark. In this figure, we filtered all GPU traces and retained only CPU timelines. The left-side of the trace represents the initialization phase, and the right-side shows the time step loop phase. The initialization phase shows that certain processes (e.g., processes on node `kid047`) have a short (~600ms) `cuInit` phase and enter `MPI_Allreduce` waiting for other processes (e.g., processes on node `kid043`) that take much longer (~7s) to finish their `cuInit` phase. Here, slow `cuInit` on some nodes is delaying all processes. When we compared 32 vs. 64 MPI-processes strong scaling LAMMPS-GPU executions in G-HPCToolkit, we observed a 21% scalability loss attributed to `MPI_Allreduce` waiting for `cuInit`, using

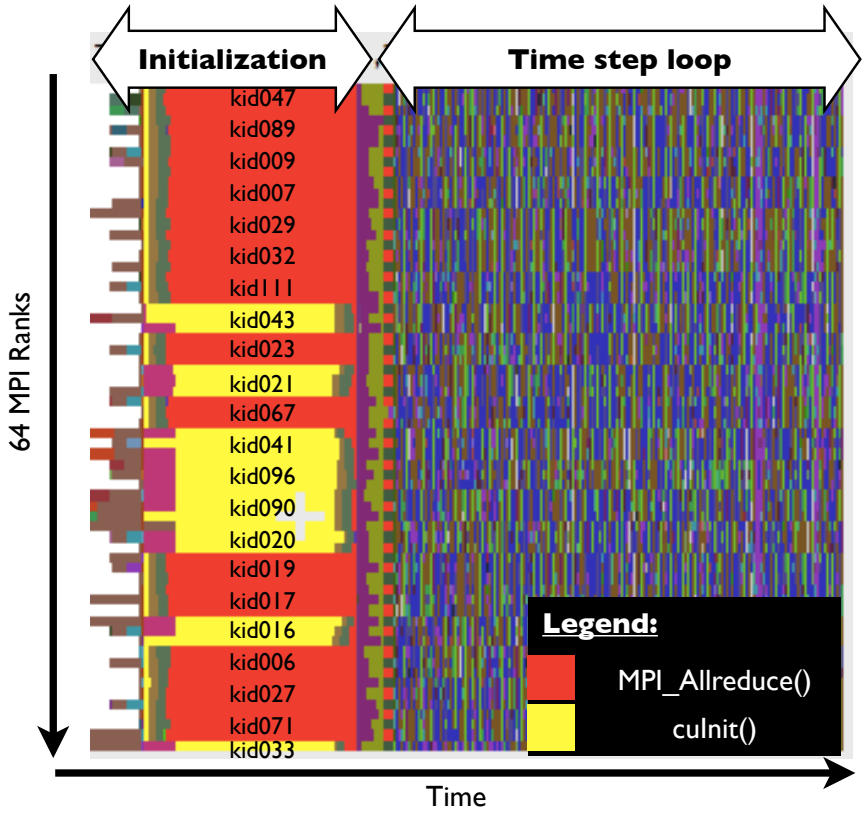


Figure 2.10 : `cuInit` delaying `MPI_Allreduce` in LAMMPS. Some nodes, such as `kid043`, are several orders of magnitude slower in performing the initialization via `cuInit` API. `MPI_Allreduce` called soon after `cuInit` causes processes to stall for late arrivers.

HPCTOOLKIT's differential profiling capabilities [53]. Repeated runs and microbenchmarks confirmed a systematic problem in `cuInit` on the same set of nodes. KIDS administrators reported that GPU power capping was enabled on some nodes. Subsequent system upgrades resolved the issue.

2.7.3 Stalls on Titan and KIDS

For our Titan case study, we again chose LAMMPS-GPU as our sample program. We ran LAMMPS-GPU on Titan using 1024 MPI ranks. Our blame-shifting idleness analysis showed no glaring algorithmic problems, but it weakly hinted at a problem with `MPI_Allreduce`. To investigate that problem, we looked at the lightweight traces.

Figure 2.11 highlights a typical iteration (one of many) that suffered an unexpected

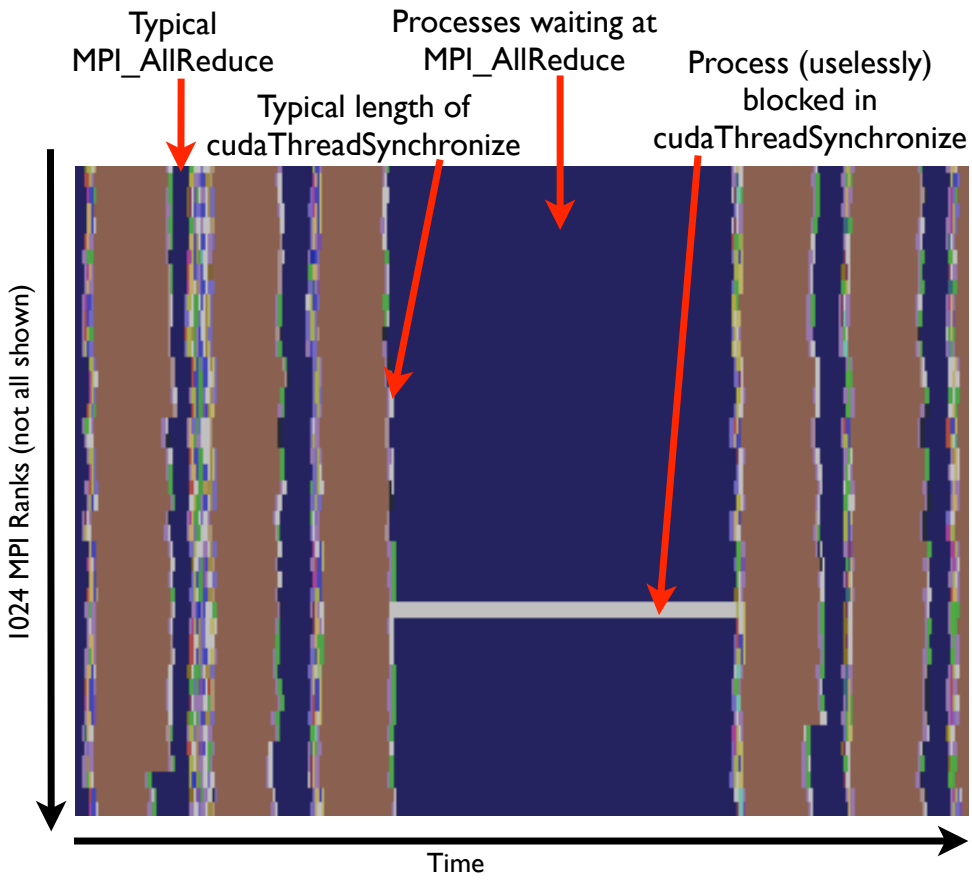


Figure 2.11 : Blocking of CUDA synchronize operations on Titan in LAMMPS. Sporadically, some processes take unusually long time to return from a blocking CPU-GPU synchronization call. An `MPI_Allreduce` following the blocking synchronization call is delayed on all processes because of a few late arrivers.

delay. Visual inspection of the problematic `MPI_Allreduce` instances showed that one or two of the preceding CUDA synchronize operations took a substantial time. Since a single rank behaving poorly can sabotage an MPI collective operation, we focused on the suspicious CUDA synchronize operations. At first, we suspected a hardware problem. This hypothesis, however, was easily ruled out as it was never the same MPI rank behaving badly. Further inspection of the trace data confirmed that the CPU samples were sparse during the offending stalls. The absence of samples typically indicates a system call that blocked. Our stall analysis confirmed this. Furthermore, by inspecting the GPU traces, we noticed the GPU activity to which the CPU was synchronizing, had already completed. Hence, the extended waiting time was a waste. Our stall analysis also quantified the cumulative delay. While the

Num MPI ranks	128	256	512	1024
Time wasted in OS stalls	1.3%	1.9%	18.4%	17.4%

Table 2.3 : Time wasted in OS stalls in LAMMPS on Titan supercomputer.

delay was relatively minor for each instance of `MPI_Allreduce`, the cumulative slowdown was about 17%.

The next question we considered was the scaling consequences of the phenomenon. The study shown in Table 2.3 indicates a scaling problem. There is a significant jump when moving from 256 MPI ranks to 512 MPI ranks.

The next step in tracking down causes was to run LAMMPS-GPU on KIDS to see if we could observe the phenomenon on a different platform. On KIDS, for a 128-rank configuration, we observed the analogous stall again, but the cumulative time wasted showed more variation — between 10% and 30%.

Since the blocking stalls occur on two separate platforms, we next questioned whether this phenomenon was peculiar to LAMMPS-GPU. To answer this question, we constructed a "proxy" app that just launched some GPU kernels, called CUDA synchronize, and then did a reduction. The proxy app showed the same random blocking stalls exhibited by LAMMPS-GPU.

Next, we measured the proxy app *directly* on KIDS, using Intel x86 `rdtsc` instructions. For each iteration of the proxy test loop, we measured the time of CUDA synchronize operation on each of the 128 nodes. The results of this test showed that many iterations had one or two nodes with exceptionally long synchronization periods. In addition, when comparing different iterations that had outliers, the MPI rank was not the same. This is exactly the same pattern as revealed by G-HPCToolkit. Therefore, we concluded that our tool was not the source of the anomaly.

Given the nature of the blocking stalls, our intuition was that the synchronization strategy employed by the CUDA runtime/GPU driver combination might be suboptimal. Fortunately, CUDA has an API call (`cudaSetDeviceFlags`) for configuring the waiting strategy employed by a CUDA synchronization operation. A CPU can either yield-wait, spin-wait, or combined-yield-spin-wait for a GPU activity. The default CUDA runtime synchronization strategy is

to spin-wait for a while and then yield-wait. On KIDS, we changed the default strategy to use exclusively spin-wait. Unfortunately, this change did not address the sporadic blocking phenomenon. Observing CUDA synchronization calls under `strace` revealed that the spin-waiting did not eliminate `ioctl` system call. A side thread, launched by the CUDA runtime, continues to make `ioctl` system calls, and these system calls can block. A collaborator at Nvidia has filed a driver bug report based on our observations [58], but it was resolved stating "the behavior is known and expected."

At this point, we need more data. All we know is that *something* systemic is degrading the performance of applications that use CUDA synchronization operations prior to `MPI_Allreduce`. The problem does not appear to be endemic to Titan. Further investigation will be needed to determine the underlying cause.

2.7.4 Overhead evaluation

Here, we present a preliminary evaluation of the runtime overhead of G-HPCTOOLKIT on the KIDS and Titan supercomputers. The point of this preliminary evaluation was to confirm that our hybrid sampling-plus-instrumentation technique did not introduce unacceptable overhead. Our "at the terminal" intuition was favorable, but it was gratifying to see our initial impressions confirmed by a little data.

For our overhead study, we compare the overhead introduced by G-HPCTOOLKIT with that of the original unmonitored execution. We also compare G-HPCTOOLKIT's CPU+GPU monitoring overhead with that of HPCTOOLKIT's CPU-only monitoring to quantify the additional overhead introduced by our GPU performance measurement strategies. We used LAMMPS-GPU running on one CPU core utilizing one GPU for our empirical experiments. We measure our overhead for both profiling and tracing. We used the default provided `in.gpu.rhodo` LAMMPS input file, which is a rhodospin protein benchmark performing 200 time step simulation on 256000 atoms, for our experiments. We used HPCTOOLKIT's default sampling rate of 200 samples per second. Table 2.4 shows the observed monitoring overhead. G-HPCTOOLKIT's CPU+GPU monitoring overhead is about 5%, and it is in the ballpark of the original CPU-only monitoring overhead of HPCTOOLKIT for both profiling and tracing. The slightly higher overhead of G-HPCTOOLKIT happens

System	Base running time in sec	Profiling time in sec (overhead%)		Tracing time in sec (overhead%)	
		HPCToolkit	G-HPCToolkit	HPCToolkit	G-HPCToolkit
KIDS	92.948	98.116 (5.56%)	98.038 (5.48%)	97.802 (5.22%)	98.036 (5.47%)
Titan	130.352	137.193 (5.25%)	138.062 (5.91%)	137.907 (5.80%)	138.368 (6.15%)

Table 2.4 : Measurement overhead for LAMMPS-GPU on KIDS and Titan supercomputers.

because of the wrapping of CUDA API calls and insertion of events into GPU streams to measure kernel timings.

While these results are promising, they are not definitive. A detailed study of the monitoring overhead for a broad collection of applications and comparison with tools such as TAU and Vampir are outside the scope of this dissertation.

2.8 Discussion

Prior studies [22, 137, 222, 225] analyzed idleness in homogeneous processor environments. In this chapter, we showed that the blame-shifting strategy, pioneered by Tallent and Mellor-Crummey [222] for identifying the causes of idleness via sampling techniques in homogeneous architectures, is effective for assessing idleness in heterogeneous architectures as well. Blame shifting is an effective, lightweight strategy in approximately identifying the critical path in a parallel program.

While we expected to tune a few key GPU kernels using the insights gained from blame shifting, in the limited set of case studies, the converse was true—we tuned the CPU code. In addition to profiling using blame shifting, we found lightweight execution traces of CPU-GPU executions to be of immense help. It came as a surprise that our execution traces could also pinpoint malfunctioning hardware. The execution traces also enabled us to recognize the operating system blocking phenomenon, which led us to develop the stall analysis.

We need better ways to extract performance data from the operating system, especially about blocking system calls. For example, the Solaris operating system provides an accounting of time spent inside a blocked system call when profiling with `ITIMER_REALPROF` [181]. Other approaches could involve tracing kernel calls or system-wide sampling, but these capabilities are not universally available.

Hardware and runtime support for analyzing the performance of GPUs is in its infancy.

We had to devise a gamut of workarounds to overcome the limitations of current GPU hardware and software to implement the CPU-GPU blame shifting. On our behest, some accelerator vendors are considering incorporating sampling-based performance analysis support in their specifications, hardware, and runtime libraries. A continuous engagement of performance tools developers with accelerator manufacturers will be important for improving the capabilities of performance tools for heterogeneous processor architectures.

Chapter 3

Eliding Redundant Barriers

There are no constraints on the human mind, no walls around the human spirit, no barriers to our progress except those we ourselves erect.

Ronald Reagan

Once a computation is well partitioned, the next step in tuning a parallel program is to reduce the overheads of parallelism. Few real-world problems are embarrassingly parallel. Hence, concurrent execution entities in a parallel program often need to communicate and synchronization with one another. Some types of synchronization involve only a few participating processes—for example, exchange of messages between two processes. Other types of synchronization involve all participating processes—for example, reduction of a partial sum computed by all processes.

Unnecessary and *costly* synchronizations make parallel programs scale poorly. In this chapter, we discuss alleviating the overheads introduced by *unnecessary barrier synchronization* in Single Program Multiple Data (SPMD) style *distributed-memory* parallel programs. Alleviating the overheads of barrier synchronization in *shared-memory* parallel programs is outside the scope of this dissertation. In the next chapter, we discuss reducing the overhead of *costly synchronization* via efficient lock designs for achieving mutual exclusion in *shared-memory* parallel programs.

In this and all subsequent chapters, we shift our focus on to homogeneous processor architectures instead of heterogeneous processor architectures that we discussed in the previous chapter.

3.1 Motivation and overview

A *barrier* is a critical synchronization construct used in parallel programs to separate an execution into different phases. Every process executing a barrier waits until all other processes participating in the barrier arrive at the barrier. Mellor-Crummey and Scott [157] and Hoefler [90] have performed an exhaustive analysis of barrier algorithms for shared-memory and distributed-memory parallel programs, respectively. Algorithmic analysis of barriers is not topical to our study. Barriers involve system-wide communication; hence, they incur high latency and scale poorly [7, 50, 233]. Furthermore, a delay in one process to arrive at a barrier, either due to load imbalance or due to operating system stalls [43, 188], causes delays in all participating processes. Our focus in this chapter is to identify when a barrier could have been avoided and if so how. In the ensuing text in this chapter, references to a "barrier" mean such inter-process synchronization mechanism often used in SPMD programs. Other intra-process synchronization mechanisms such as a barrier among CPU threads within a same process, a memory barrier a.k.a a fence, and a barrier among multiple hardware threads within a GPU, which are homonymous with the term barrier, are not pertinent to our discussion.

Large scientific programs often use Partitioned Global Address Space (PGAS) programming paradigm. The data is distributed on participating processes in PGAS programs. Any process may access the data residing on any other process through one-sided asynchronous `get` and `put` primitives. PGAS programs use program-wide *global barriers* to ensure consistency of data after asynchronous updates from the participating processes.

A barrier may be necessary *only in some execution contexts but not all contexts.* Prodigal use of barriers in PGAS programs leads to poor scalability. Redundant barriers are surprisingly frequent in large PGAS programs due to layers of software abstractions and composition of conservatively implemented APIs. Redundant barriers in *hot* execution contexts offer an excellent opportunity for optimizing communication-intensive scientific codes. In this chapter, we discuss our experience building an application-specific dynamic optimization able to detect and skip barriers that are redundant in their runtime calling contexts. The application is the NWChem [230]—a computational chemistry code with more than six million

lines of C, Fortran77, and C++ code. NWChem is implemented atop multiple abstraction layers for data and communication. NWChem includes software abstractions for its internal tasking, load balancing, memory management, and checkpoint/restart mechanisms. The abstractions used for software modularity hinder optimizations. Specifically, communication libraries written for generic use, introduce context-sensitive redundant synchronization. Such overheads limit the scalability of NWChem. Code size, multiple programming languages, and multiple programming models, make compiler-based redundancy elimination impractical on production software such as NWChem.

To alleviate the overheads introduced by unnecessary barrier synchronization, we have designed a simple, yet effective, *runtime technique that pinpoints and automatically elides context-sensitive redundant barriers*. The crux of our automatic elision idea is to "learn from history". We identify barriers by their calling contexts (call-path) at runtime. We determine if a barrier in its calling context is *locally* redundant within a process. We infer that a barrier in its calling context is *globally* redundant if the barrier is redundant on all participating processes. Global redundancy is easy to derive by replacing a barrier with a reduction operation during a *learning phase*. Once learned, a process speculatively skips future instances of barriers in the same calling contexts where they were previously deemed globally redundant.

Since precise data dependence detection is *not* our objective, we can afford to employ a coarse-grained dependency detection mechanism that *may* report false positives.[1] By relaxing precision, we achieve *low overhead* in our analysis—an essential requirement for dynamic optimizations. We call this mechanism as an *automatic* barrier elision technique. The analysis incurs less than 1% runtime overhead. The method is guaranteed to catch bad speculation at runtime. Misspeculation can be handled either by aborting the execution or by restarting from a checkpoint. Bad speculation has not happened in our production runs. Modern scientific programs utilizing the checkpoint-restart technique make our technique even more attractive. Developer confidence in the applicability—whether for correctness or performance—however, has to be gained through testing coverage.

Besides the *automatic* barrier elision technique, we also devised a *guided* barrier elision

[1] A false positive happens when the analysis suspects data dependence when there exists none.

approach. In the guided barrier elision approach, we execute the program on several training inputs and collect a large set of calling contexts where the barriers may be redundant in an execution. We do not elide barriers during this training phase. We classify contexts into equivalence classes, where all contexts in an equivalence class share a common subcontext. Subcontexts capture the intuition that the barrier redundancy inside a library module is determined by prior barrier calls performed by its callers. Subcontexts provide useful insights about how redundancies arise in the code. We then present these subcontexts as elision candidates for the inspection by a developer. The set of subcontexts approved by the developer become inputs for production-time barrier elision. The guided approach does away with speculation and builds developer confidence in the elision process.

Both automatic and guided techniques were able to elide up to 45% and 63%, respectively, of all barriers encountered during NWChem science production runs. Eliding barriers resulted in end-to-end application runtime improvements up to 14% when running on 2048 cores. In addition, our approaches provide valuable tools for program understanding. We uncovered several context-insensitive redundant barriers, which we then removed from the source of NWChem.

3.2 Contributions

This work, which appeared in the Proceedings of 20^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming [40], makes the following contributions.

1. It presents a context-sensitive, dynamic program analysis able to pinpoint, quantify, and elide redundant barrier synchronization in PGAS programs,

2. It applies the barrier elision technique to the NWChem computational chemistry code and demonstrates that redundant barriers are surprisingly frequent (63%) during its science production runs,

3. It demonstrates end-to-end speedup as high as 14% at 2048 cores on production scientific runs of NWChem, and

4. It provides valuable insights to application developers on context-sensitive as well as context-insensitive redundant barriers in NWChem.

3.3 Chapter roadmap

The rest of the chapter is organized as follows. Section 3.4 describes the aspects of synchronization in modern scientific codes and Section 3.5 provides the necessary properties of safe barrier elision. Section 3.6 sketches the automatic barrier elision technique, while Section 3.7 presents the guided barrier elision technique. Section 3.8 highlights the structure of NWChem together with implementation details of our infrastructure. Section 3.9 provides the insights we gained about NWChem via our techniques. Section 3.10 presents an empirical evaluation with further discussion in Section 3.11.

3.4 The problem with synchronization

Large scientific applications [88, 211] employ a hierarchy of libraries to implement layered abstractions. In the absence of contextual knowledge, libraries are designed for generality in such a way that any parallelism is quiesced upon entry and exit from the respective module. As tracking individual dependencies is challenging, the synchronization usually involves heavyweight operations such as barriers and fences. For example, ScaLAPACK [20] API calls use collective synchronization semantics, which may hinder the overall application scalability. Application programmers may also add synchronization to ensure semantic guarantees when employing libraries for asynchronous operations. In cases where a lower-level library routine already provides stronger synchronization guarantees than advertised (sometimes undocumented) in its interface specification, the ensuing communication redundancy causes non-trivial performance overheads.

Figure 3.1 is an anecdotal code where transitions from higher-level abstractions to lower-level abstractions can sometimes cause redundant barriers. In this figure and the following text, we symbolize a caller to a callee relationship as `caller()`→`callee()`. The `ClientA()`→`MatMul()` transition has no redundancy since the `ClientA()` relies on the barriers provided by the `MatMul()` function in the lower-level library. The `ClientB()`→`MatMul()`

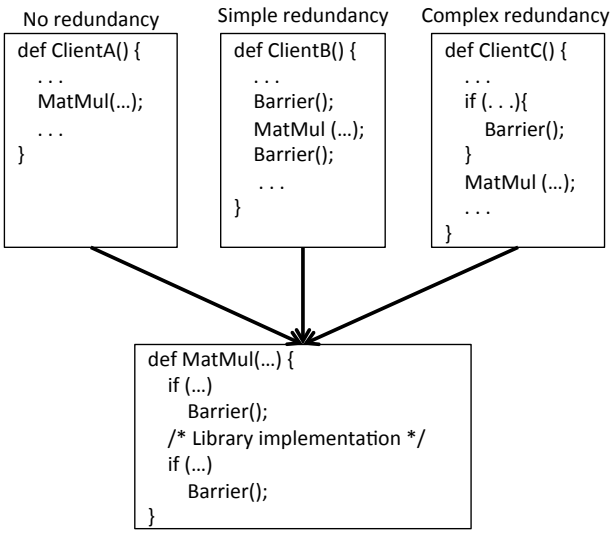


Figure 3.1 : Composition of synchronized API calls.

and `ClientC()`→`MatMul()` transitions can make some barriers redundant, and it is progressively more complex to detect or eliminate such redundancies. Eliminating barriers in the `MatMul()` function breaks the code in the `ClientA()`, whereas keeping barriers in the `MatMul()` function causes redundancy in other clients. As shown later in Section 3.9 this is a common occurrence in NWChem.

Static program analysis [65, 109, 149, 245] has been successfully used for synchronization optimizations, and it may be able to handle our example. Static analysis, however, faces great engineering challenges when dealing with the characteristics of existing full-fledged HPC applications which: 1) use combinations of multiple languages, such as C/C++/Fortran; 2) are written in a modular fashion with calls into manifold "libraries"; and 3) are built on layers with different semantic abstractions. While language designers [38, 45, 48] and application developers are striving to expose concurrency inside an application, software engineering practices (modularity, composability) and development tools (multiple compilers) are busy negating it.

Over-conservative synchronization already appears in the current generation of HPC codes. This pattern is likely to be pervasive in the next generation of codes designed for extreme scaling. As the scientific community is moving towards multiphysics multi-scale simulations, HPC codes are universally refactored as compositions of parallel libraries, solvers, and runtimes. The next generation of codes that employ Domain Specific Languages

(DSL) [123, 216] or high-productivity languages such as Chapel [45] will exhibit similar characteristics, as their compilers use source-to-source translation with calls into libraries implementing the language-level abstractions. In these cases, statements are compiled mostly independently from one another into complicated hierarchies of parallel calls.

Reasoning about synchronization is challenging in codes that use Partitioned Global Address Space (PGAS) abstractions and one-sided Remote Direct Memory Access (RDMA) [45, 48, 171, 172] based communication. Memory in the PGAS model is classified as either private or shared. Private memory can be accessed only by a single task. Shared memory can be accessed by any task using load/store instructions or RDMA operations. Unlike MPI, where `send` and `receive` pairs couple data transfers and synchronization, in PGAS the two are decoupled.

The optimizations hereinafter are designed to eliminate redundant barrier operations in the NWChem computational chemistry code described in Section 3.8. NWChem combines PGAS and RDMA concepts; the scientific community foresees that future codes will employ at least one of these mechanisms.

3.5 Reasoning about barrier elision

A barrier may be redundant if there are no data dependence edges (*read-write, write-read, or write-write*) originating from before the barrier on one task and terminating after the barrier on another task. Figure 3.2(a) shows a barrier necessary to resolve a write-read conflict. Figures 3.2(b)- 3.2(e) show several cases of redundant barriers with no data dependence between processes across barriers. A data race detector finds dependencies by tracking individual addresses accessed by the processing elements (PEs) before a barrier and comparing these with the accesses after the barrier.

In practice, address tracking for data race detection may incur higher costs than barrier elision could hope to gain. For example, Park et al. [185] describe a precise data race detector using dynamic analysis for PGAS programs with one-sided communication, which incurs 50% overhead at 2000 cores. Their technique is directly applicable here, but its runtime overhead is higher than the total time spent by NWChem in barriers (up to 20%) in our experiments.

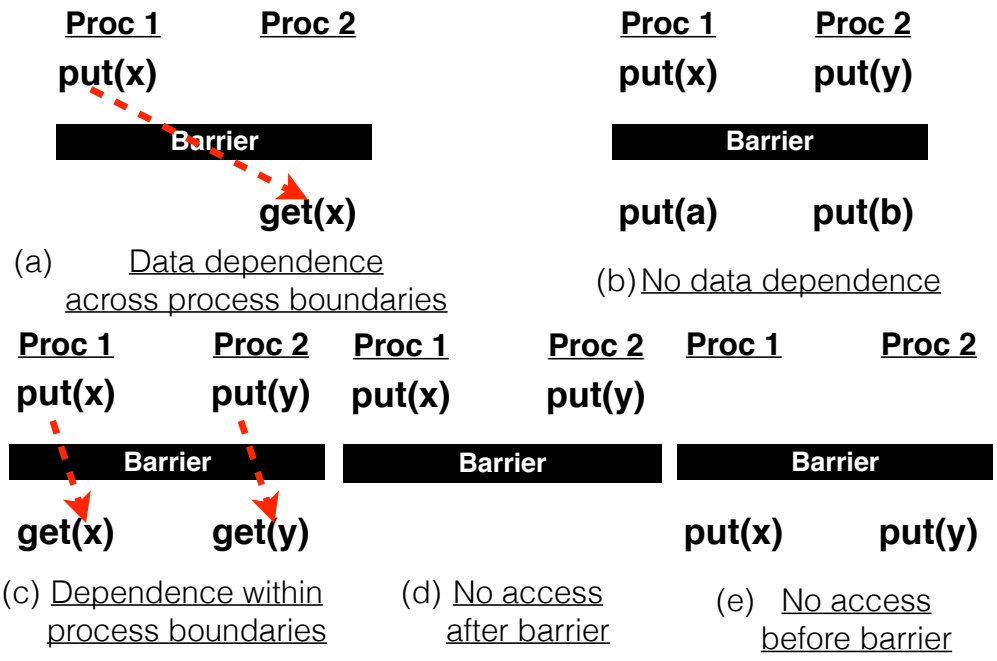


Figure 3.2 : Data dependence and barrier redundancy

Memory accesses in PGAS models can be distinguished as follows:

- N - access to memory that is private to a processing element (PE) and cannot be accessed by any other PE. The term N stands for "**N**o" shared memory access.
- L - access to shared memory that has an affinity with one PE, which can perform load/stores into it. Access from any other PE involves RDMA or other calls into the runtime. The term L stands for "**L**ocal" shared memory access.
- R - access to shared memory that is remote and can be accessed only using RDMA. The term R stands for "**R**emote" shared memory access.

To achieve lower overhead, we over-approximate the dependencies by assuming that any access (L,R) of shared data *before* the barrier can alias with any (L,R) shared data access *after* the barrier. Furthermore, we do not distinguish a write from a read, and treat any access as a write operation. Because precise knowledge of individual addresses is no longer needed, this assumption reduces the overhead of instrumenting each memory access and simplifies analysis. We only need to intercept access to shared data.

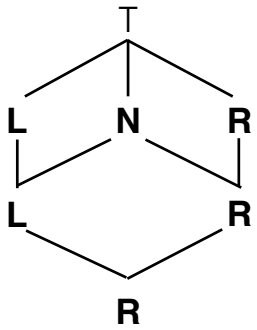


Figure 3.3 : Lattice of memory access types.

In the following text, the term "barrier", denoted by **B**, refers to a dynamic instance of a barrier operation, regardless of its source code location. We associate a memory access summary value (S_i) with any portion of the program executed between two barrier operations. This access summary has one of the (N,L,R) values, and it is computed as the transitive closure of the types of all memory accesses in that particular code region on i^{th} PE. The access summary is computed using the meet operation described below.

Definition 3.1 (ACCESS: $\bigwedge$) *N, L, and R form a monotonically descending lattice (Figure 3.3), where the meet operation ($\bigwedge$) between two types of accesses is defined as follows:*

$$
\begin{aligned}
N \bigwedge N &= N \\
N \bigwedge L = L \bigwedge N &= L \\
N \bigwedge R = R \bigwedge N &= R \\
L \bigwedge L &= L \\
L \bigwedge R = R \bigwedge L &= R \\
R \bigwedge R &= R
\end{aligned}
$$

Intuitively, this provides a hierarchy of observability for memory accesses. Remote operations (R) take precedence over all other types. Operations on shared data with a local affinity (L) take precedence over access to private data (N).

3.5.1 Ideal barrier elision

Given any execution trace $S_i^b \mathbf{B} S_i^a$, where S_i^b and S_i^a are the memory access summaries of task i before and after the barrier **B**, we compute a trace with global access summaries as

Rule	Trace	Transformation	Memory access summary
1	$N\mathbf{B}N$	$N\not{\mathbf{B}}N$	$N \bigwedge N = N$
2	$N\mathbf{B}L$	$N\not{\mathbf{B}}L$	$N \bigwedge L = L$
3	$N\mathbf{B}R$	$N\not{\mathbf{B}}R$	$N \bigwedge R = R$
4	$L\mathbf{B}N$	$L\not{\mathbf{B}}N$	$L \bigwedge N = L$
5	$L\mathbf{B}L$	$L\not{\mathbf{B}}L$	$L \bigwedge L = L$
6	$L\mathbf{B}R$	$L\mathbf{B}R$ (none)	R
7	$R\mathbf{B}N$	$R\not{\mathbf{B}}N$	$R \bigwedge N = R$
8	$R\mathbf{B}L$	$R\mathbf{B}L$ (none)	L
9	$R\mathbf{B}R$	$R\mathbf{B}R$ (none)	R

Table 3.1 : The rules that dictate allowable transformations, given an observed execution trace $S^b\mathbf{B}S^a$. The new memory access summary comes into effect when the transformation is applied.

$$\underbrace{N\mathbf{B_1}L}_{\text{Rule 2}}\mathbf{B_2}N\mathbf{B_3}R$$
$$N\not{\mathbf{B_1}}\underbrace{L\mathbf{B_2}N}_{\text{Rule 4}}\mathbf{B_3}R$$
$$N\not{\mathbf{B_1}}L\not{\mathbf{B_2}}\underbrace{L\mathbf{B_3}R}_{\text{Rule 6}}$$
$$N\not{\mathbf{B_1}}L\not{\mathbf{B_2}}L\mathbf{B_3}R$$

Table 3.2 : Example application of rules from Table 3.1.

$S^b\mathbf{B}S^a$, where $S^b = \bigwedge_{i=0}^{Procs} S_i^b$ and $S^a = \bigwedge_{i=0}^{Procs} S_i^a$.

For any sequence $S^b\mathbf{B}S^a$, there can be nine different orderings of global access summaries—shown in column 2 of Table 3.1. For each combination, we can decide whether the barrier is redundant based on the observability of the access summary. When a barrier is deleted, the global access summaries before and after it need to be combined into a single value, $S^b \bigwedge S^a$. If the barrier is retained, the access summary before the barrier has no relevance to what ensues after the barrier and hence remains S^a. As Table 3.1 shows, in six of the nine variations, the barrier can be safely eliminated. Intuitively, any barrier preceded by remote accesses R is retained; any barrier that neighbors purely local accesses N as well as barriers surrounded by shared accesses with local affinity L can be elided.

Given a trace, the rules can be applied in any order. Considering a sample trace $N\mathbf{B_1}L\mathbf{B_2}N\mathbf{B_3}R$, Table 3.2 shows a sequence of valid barrier elision transformations.

While providing good opportunity for barrier elision, there are several challenges to implementing this approach:

1. The transformations require inspecting the access summary both *before* and *after* the barrier.

2. The transformation performed at one barrier affects the resulting access summary after the barrier, hence the transformation possible at the subsequent barrier. In the previous example, applying Rule 4 at barrier B_2 changes the access summary after B_2 from N to L; consequently, at B_3, one can't use Rule1 $N\mathbf{B_3}R \Longrightarrow N\not{\mathbf{B_3}}R$. Thus, an optimal barrier deletion algorithm *requires processing the whole program execution trace.*

3. Knowing the system-wide access summary at a barrier *requires a communication with all processes*, which defeats the purpose of deleting the barrier.

3.5.2 Practical barrier elision

For practical reasons, we adopt a simplified approach that uses *only* information about the memory access summary *before* a barrier in a manner that is independent of the order of the barrier elision. From Table 3.1 the following is apparent:

1. Any barrier preceded by purely local accesses (global access summary N) can be elided.

2. Elision of barriers preceded by purely local access does not affect the "redundancy" of any barriers that follow.

Indeed, consider the execution trace $N\mathbf{B_1}X\mathbf{B_2}Y$, where $X, Y \in \{N, L, R\}$. After the $N\not{\mathbf{B_1}}$ transformation, the access summary before $\mathbf{B_2}$ is $N \bigwedge X = X$. Hence, deleting $\mathbf{B_1}$ has no effect on the access summary before $\mathbf{B_2}$. Consequently, the transformations possible at $\mathbf{B_2}$ are unaffected by the transformation performed at $\mathbf{B_1}$.

We summarize our simplified transformation rules in Table 3.3. We apply the simplified transformation to the earlier example in order in Table 3.4.

These rules capture three out of the previous six cases where elision is possible and perform well in practice for NWChem. Some codes using PGAS paradigms but optimized for locality may have barriers surrounded by L summaries, i.e. accesses in the global space but with local affinity. Our simplified approach will classify these barriers as necessary. Note that NWChem developers optimized some of these cases explicitly.

The final issue is that of the necessity of performing communication to compute the global access summary before a barrier. We identify a barrier by its call-path, a.k.a. *calling*

Rule	Trace	Transformation
1	$N\mathbf{B}$	$N\cancel{\mathbf{B}}$
2	$R\mathbf{B}$	$R\mathbf{B}$ (none)
3	$L\mathbf{B}$	$L\mathbf{B}$ (none)

Table 3.3 : Simplified rules used in our implementation.

$\underbrace{N\mathbf{B_1}}\ L\mathbf{B_2}N\mathbf{B_3}R$
Rule 1
$N\cancel{\mathbf{B_1}}\ \underbrace{L\mathbf{B_2}}\ N\mathbf{B_3}R$
Rule 3
$N\cancel{\mathbf{B_1}}L\mathbf{B_2}\ \underbrace{N\mathbf{B_3}}\ R$
Rule 1
$N\cancel{\mathbf{B_1}}L\mathbf{B_2}N\cancel{\mathbf{B_3}}R$

Table 3.4 : Example application of rules from Table 3.3.

context. We assume that *a barrier that is repeatedly redundant in a calling context is likely to remain redundant for the rest of the execution under the same calling context.* This behavior forms the basis of our *automatic* elision technique described in detail in Section 3.6.

Alternatively, the program behavior can be observed for entire training executions at a small scale; subsequently, barriers that are always redundant in some calling contexts can be identified in a postmortem analysis. A production run can elide a barrier whenever its calling context matches that of the designated preprocessed calling contexts. This approach forms the basis of our *guided* analysis discussed in detail in Section 3.7.

Because of this speculative elision, our techniques work well for the repeatable behavior present in indirect HPC solvers. For other programs, a future iteration or a production run might behave differently than what has been learned for a calling context. In these cases, our approach is guaranteed to catch and report the misspeculation.

3.6 Automatic barrier elision

The automatic barrier elider works in two phases: it learns about barriers within their full calling contexts and then speculatively elides those deemed redundant (see Algorithm 1).

Algorithm 1: Automatic barrier elision

```
   Input: p = self /* implicit process identifier                                   */
   Result: SUCCESS for elided barrier or return value from a participated barrier
 1 enum {PARTICIPATE=0, SKIP=1, LEARNING=THRESHOLD+SKIP, CB=LEARNING+1}
 2 /* ctxtId is same as H(c) in the prose.                                          */
 3
 4 ctxtId = Hash(GetCallingContex())
 5 /* dict is a dictionary of <context hash, memorized transformation>             */
 6
 7 if ctxtId ∉ dict then
 9     /* First visit to this barrier                                              */
10     val = MinReduce(in S_p^b, out S^b, ...)
11     if S^b == N then
12         dict[ctxtId] = LEARNING
13     else //S^b ≠ N
14         dict[ctxtId] = PARTICIPATE
16         /* reset local state                                                    */
17         S_p^b = N
18     return val
19 if S_p^b == N then
20     switch dict[ctxtId] do
21         case SKIP
22             return SUCCESS
23         case PARTICIPATE
24             return MinReduce(in N, ... )
25         otherwise
26             /* Learning                                                         */
27             val = MinReduce(in N, out S^b, ...)
28             if S^b == N then
30                 /* When dict[ctxtId] reaches SKIP, we will start eliding      */
31                 dict[ctxtId] −−
32             else // S^b ≠ N
33                 dict[ctxtId] = PARTICIPATE
34             return val
35 else // S_p^b ≠ N
36     switch dict[ctxtId] do
37         case PARTICIPATE
38             val = MinReduce(in S_p^b, ...)
39         case SKIP
41             /* Breaking consensus, Optimistic                                   */
42             val = MinReduce(in CB, out S^b,... )
43             if S^b ≠ CB then
45                 /* Not all PEs broke consensus.                               */
46                 Report misspeculation.
48             /* else, All PEs broke the consensus, program is safe.            */
49         otherwise
50             /* Veto in skipping                                                 */
51             val = MinReduce(in S_p^b, out S^b, ...)
52             dict[ctxtId] = PARTICIPATE
54     /* reset local state S_p^b                                                  */
55     S_p^b = N
56     return val
```

We identify each barrier **B** by its calling context c, represented as $\hat{B}_c$. We encode the calling context as a hash, $H(c)$, formed from the call chain starting from `main` to the current barrier inclusive of all functions[2] along the path. The algorithm maintains a monotonically increasing barrier sequence number, incrementing on encountering each barrier, to distinguish between different barrier episodes.

During the learning phase, we observe and remember if a barrier is redundant in its calling context across all participating processes. Our algorithm *replaces the barrier with a reduction*[3] to compute the global access summary. The reduction performs a *min* operation $S^b = \bigwedge_{p=0}^{Procs} S_p^b$ on the local access summaries (S_p^b) before the barrier. A global access summary result can be either R or N.

- If R, then each PE memorizes $\hat{B}_c$ as necessary, and learning stops for that barrier; *any barrier deemed necessary once will remain classified as necessary.*
- If N, then each PE records $\hat{B}_c$ as a candidate for elision, and learning continues. If subsequent *threshold* number of visits to $\hat{B}_c$ also result in the N state, then $\hat{B}_c$ is promoted to an elidible barrier; *any barrier once classified as redundant will be skipped for the rest of the execution.*

Speculation can fail when one or more PEs violate their memory access behavior (i.e., performs an L or R local access) before arriving at a barrier promoted for elision. Our implementation, however, detects any misspeculation and then either recovers (if possible) or aborts the execution (otherwise). If checkpointing is enabled, instead of aborting, we can restart all processes from the last checkpoint, this time not eliding the barrier in any PE. The implementation continues to maintain up-to-date global access summaries between barriers and piggybacks the summary for elided barriers onto the barriers still executed. If a PE decides that a barrier previously elided became necessary, there are two options:

---

[2]Return address of the callee, to be specific.

[3]On current systems, the cost of a reduction is comparable to a barrier, if the message sent is small (eight bytes in our case) and the reduction operation is short (less than 10 instructions in our case).

1. *Pessimistic:* Record misspeculation (and either abort or restart from last checkpoint).

2. *Optimistic:* Assume that all PEs detect the same broken consensus, allowing all to participate in the barrier and maintain the integrity of the system.

We take the optimistic approach and replace the barrier with a *min* reduction with a special status CB, representing the fact that the consensus is broken. The meet operation is augmented such that $CB \bigwedge CB = CB$, $CB \bigwedge R = R$, $CB \bigwedge N = N$. If the resulting global access summary after the reduction is also CB, it means that all PEs broke the consensus and each one participated in the synchronization. If such is the case, we have two options:

1. *Pessimistic:* Locally downgrade the context $\hat{B}_t$ as necessary in all PEs to avoid future consensus breaks.

2. *Optimistic:* Assume the pattern will hold and expect future instances of $\hat{B}_t$ to be skippable or equally recoverable, and thus retain the barrier as redundant.

We again take the optimistic approach. Finally, a subtle case arises when some PEs break consensus for a specific barrier and wait in a reduction operation with the CB state whereas other PEs skip that barrier, advance, and break consensus in a future barrier. This scenario leads to two mismatching reduction operations and possible incorrect behavior or even deadlock. To handle this case, each PE passes the aforementioned unique barrier sequence number in each reduction operation; if the sequence numbers do not match, we record the misspeculation.

Training threshold: The duration of the learning phase is tunable. Shortening the learning phase might increase the number of barriers classified as redundant, but it might also increase the chance of misspeculation. In the current implementation, we use a static threshold determined through testing. The value can also be selected at runtime by developers. We discuss more details about learning in the context of NWChem in Section 3.10.

For an arbitrary application, the suggested method to choose the right training threshold is first to run the application on training inputs. During training runs, we log the decisions for all the barriers in the program without performing the barrier elision. Then, we analyze the logs offline to determine the upper bound on repetitions before a barrier stabilizes. We

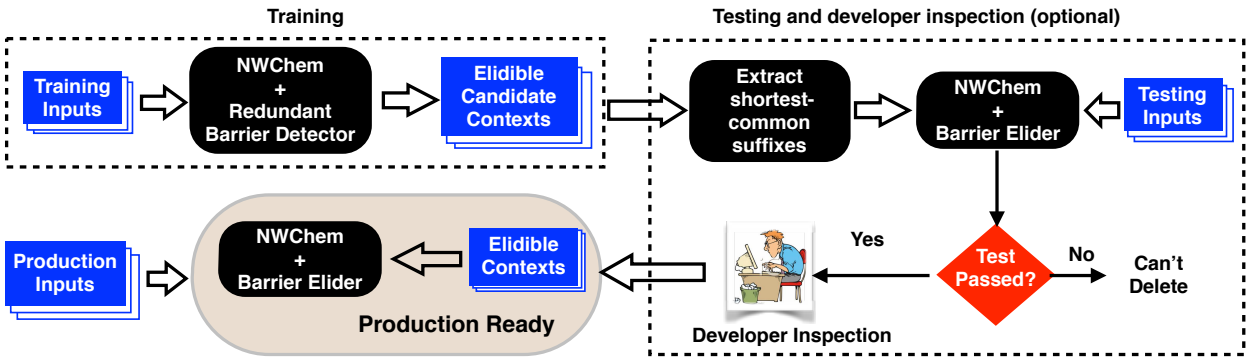


Figure 3.4 : Workflow of guided barrier elision.

can use this threshold for the actual production runs where the automatic barrier elision will be performed.

We have not encountered misspeculation in our experiments. NWChem contains a checkpoint-restart feature. We plan to blacklist misspeculated barriers and restart the execution from a checkpoint that was previously deemed safe.

3.7 Guided barrier elision

Figure 3.4 presents the workflow for a guided procedure that separates learning about barrier redundancy into an offline training. The training inputs are small-scale representatives of various real inputs. After learning, the redundant contexts are classified, validated, and approved for consumption by a *barrier elider* module for production runs. The guided analysis has different characteristics from the automatic transformation in the following ways.

1. It increases the chance of deleting the barriers missed during automatic learning.

2. It provides developers with an opportunity to inspect redundant contexts.

3. It increases developer confidence by inducing a validation step.

We start with a set of training inputs and run a **R**edundant **B**arrier **D**etector (RBD) alongside the execution. Currently, we use the same detector from the automatic analysis, but note that this can be replaced with any other race detector, even a more imprecise one. The RBD observes each barrier in its full calling context, classifying it as either redundant (if

all operations preceding the barrier are always private) or necessary (if at least one operation preceding the barrier was not private). RBD logs *all* barrier contexts for each test input.

We then perform an offline analysis, where we merge all **R**edundant **B**arrier **C**ontexts from all training runs into a single set (RBC) and all **N**on-**R**edundant **B**arrier **C**ontexts into another set ($NRBC$). In this phase, we classify a barrier in a calling context as redundant if it appears in the set of redundant barrier contexts but not in the set of non-redundant barrier contexts (in any inputs). This classification forms the set of **E**lidible **C**andidate **C**ontexts (ECC). $ECC = RBC - NRBC$.

Subcontext classification: We classify the contexts in ECC based on the intuition that in modular software paths through a certain library module are probably determined by initial conditions set by the module's callers. In this case, there may be paths local to the module where a barrier is always redundant, and there may be callers that always exercise these. This behavior is captured by the common parts of the calling context leading to redundant barriers, referred to as *subcontexts*.

Consider the example in Figure 3.5, which shows a *bottom-up view* of various call-paths that lead to the barrier B. We note that because of the bottom-up view, the direction of arrows in the figure go from a callee to a caller. In the prose, however, we use the more intuitive direction of arrows, i.e., from a caller to a callee. `Module 1` always forces a barrier before calling `Module 4`, which eventually calls a barrier—represented by the calling context suffix $N{\rightarrow}B$. Barriers called through myriad call-paths all sharing the suffix $M{\rightarrow}N{\rightarrow}B$ are redundant. However, the same is not true for `Module 2` and `Modules 3`, which do not enforce a barrier when calling a barrier through context suffix $N{\rightarrow}B$. This observation provides the insight that the call-path suffix $M{\rightarrow}N{\rightarrow}B$ causes a redundant barrier, whereas the suffix $N{\rightarrow}B$ alone is not redundant. Inspecting myriad redundant full call-paths such as $\{main{\rightarrow}\cdots W{\rightarrow}M{\rightarrow}N{\rightarrow}B\}$,$\cdots$, $\{main{\rightarrow}\cdots Z{\rightarrow}M{\rightarrow}N{\rightarrow}B\}$ is tedious and cannot provide this type of insight.

Based on the aforementioned observation, our subcontext classification employs the Algorithm 2 to find the *shortest common suffix* of a context inside all redundant barriers. The algorithm groups redundant barrier contexts in equivalence classes such that each class

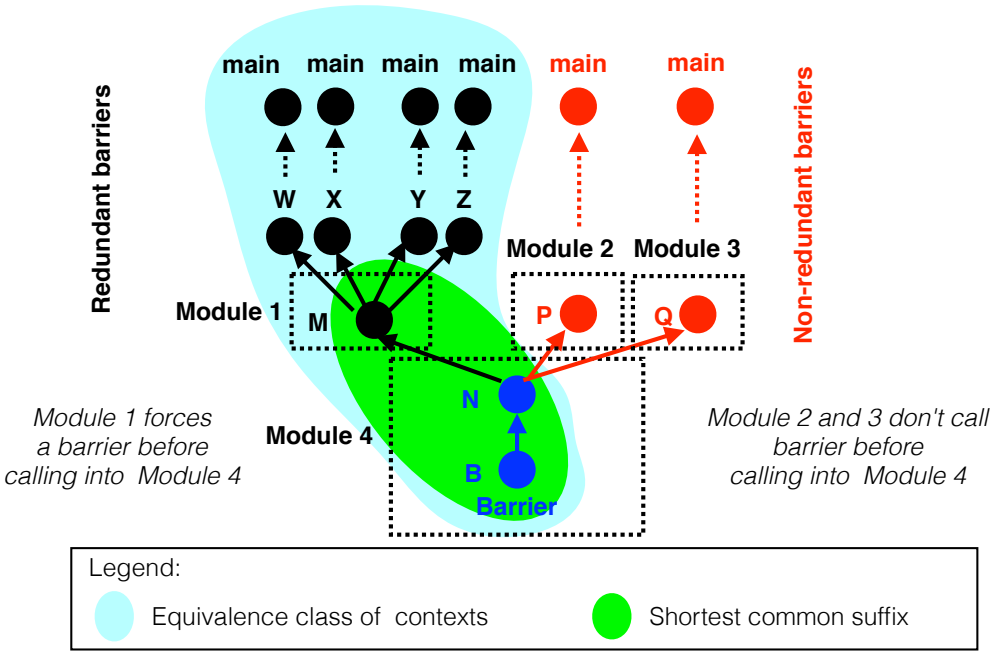


Figure 3.5 : A bottom-up view of call-paths leading to barriers. All call-paths that end with the suffix $M \rightarrow N \rightarrow B$ lead to redundant barriers.

can be represented by a shortest common suffix. The *distinguishing* shortest common suffix meets the following two criteria:

1. It does not appear as a suffix of any contexts in the non-redundant barriers set ($NRBC$). This criterion ensures that the barrier is *redundant*.

2. The suffix of length one less is present in the ($NRBC$) set. This criterion ensures that the *distinguishing* suffix is the *shortest* to classify the subcontext as redundant.

The algorithm terminates since ECC monotonically decreases in size reaching $\emptyset$, when the length of suffix equals the maximum length context. In the aforementioned example, the call-paths $\{main \rightarrow \cdots W \rightarrow M \rightarrow N \rightarrow B\}, \cdots, \{main \cdots Z \rightarrow M \rightarrow N \rightarrow B\}$ will be classified into one equivalence class represented by the common suffix $M \rightarrow N \rightarrow B$.

The output of this stage is the **E**lidible **C**andidate **S**uffix set (ECS), which contains subcontexts necessary to designate a barrier as redundant.

Test validation: The test module orders the sets in ECS by their cardinality, i.e. the number of redundant barriers that share a particular subcontext, picks the largest one, and

Algorithm 2: Barrier suffix context classifier

```
   Input: NRBC/*Non-redundant Barrier Contexts*/
   Input: RBC/*Redundant Barrier Contexts*/
   Output: ECS/*Elidible Context Suffixes*/
 1 ECC = RBC-NRBC /*Elision candidate contexts*/
 2 ECS = ∅
 3 len = 1
 4 while ECC ≠ ∅ do
 5 |   /*Find suffixes of length=len*/
 6 |   A = {Suffix (C, len) ∀ C ∈ ECC}
 7 |   B = {Suffix (C, len) ∀ C ∈ NRBC}
 8 |   /*Find elidible suffixes of length=len*/
 9 |   S = A-B
10 |   ECS = ECS ⋃ S
11 |   /*Remove classified contexts from ECC*/
12 |   ECC = ECC - {∀ C ∈ ECC s.t. Suffix (C, len) ∈ S}
13 |   len++
14 return ECS
```

starts extending the set including one subcontext at a time to test. The testing employs a *barrier elider* module that elides barriers whose runtime calling contexts match the subcontexts being tested. Test validation cross-checks against the expected results.

We also present the sorted *ECS* and the testing results to the developer for further investigation. The developer then has the option to further filter the contexts based on intuition.

Production-time barrier elision: The developer-approved contexts become inputs to production runs. The production runs use a barrier elider module to skip barriers whose contexts match the developer approved elidible contexts. We present the details of insights gained from our technique in Section 3.9. These production runs have the same safety guarantees of identifying misspeculation and the possibility of restarting as the automatic algorithm already discussed in Section 3.6.

3.8 Barrier elision in NWChem

In this section, we provide the scientific details of NWChem, its code structure, and our strategy for minimally instrumenting NWChem. We also discuss our technique of collecting and maintaining calling contexts and portability of our techniques to other applications.

3.8.1 NWChem scientific details

NWChem [230] is a computational chemistry code widely used in the scientific community. NWChem provides a portable and scalable implementation of several Quantum Mechanics and Molecular Mechanics methods: Coupled-cluster (QM-CC), Hartree-Fock (HF), Density functional theory (DFT), Ab initio molecular dynamics (AIMD) etc. The results in this chapter are for high-accuracy QM-CC, where many of the NWChem computational cycles are spent. QM-CC is by far the most computationally intensive, and therefore, the most scalable method in NWChem. The other methods have increasing demand for global synchronization, such as the HF method that generates starting vectors for QM-CC. As we move along to the methods that demand more synchronization, we expect the optimizations developed in this chapter to deliver even better performance gains. We used the following two important science production runs to evaluate the benefits of barrier elision:

DCO: Accurate simulation of the photodissociation dynamics and thermochemistry of the dichlorine oxide (Cl_2O) molecule. This simulation is important for understanding the catalytic destruction of ozone in the stratosphere, where this molecule plays the role of reaction intermediate.

OCT: Simulation of the thermochemistry and radiation absorption of the oxidized cytosine molecule ($C_4H_6N_3O_2$). This simulation is key to understanding the role of oxidative stress and free radical-induced damage to DNA.

Both runs first perform a HF simulation to obtain starting vectors. Subsequently, each run performs a different type, but algorithmically similar, QM-CC simulation.

3.8.2 NWChem code structure

NWChem contains more than six million lines of code written in C, C++, Fortran and Python. Besides the chemistry solvers (written in Fortran, C++, and Python), it contains a complicated runtime infrastructure (in C) that implements support for tasking, load balancing, memory management, resiliency, communication, and synchronization. Communication and synchronization in NWChem are handled across multiple software modules. The Global

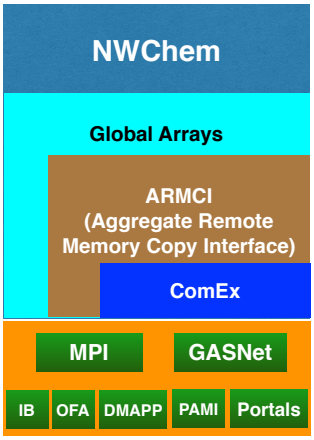


Figure 3.6 : Layers of abstractions in NWChem software stack.

Arrays [172] (GA) framework provides shared memory array abstractions for distributed memory programming with primitives such as `get` and `put` on array sections (Figure 3.6). GA is implemented atop other communication libraries such as Aggregate Remote Memory Copy Interface (ARMCI) [171] and Communication runtime for Exascale (ComEx) [54]. These libraries make the Global Arrays layer portable by hiding the low-level RDMA and synchronization primitives. Any of these layers, however, may directly access lower-level communication libraries such as MPI. ARMCI and ComEx are implemented atop native communication APIs such as Cray DMAPP [57], InfiniBand Verbs [95], and IBM PAMI [121], among others.

NWChem has some hand optimizations to elide unnecessary synchronizations. Such cases are present in situations where all tasks perform local operations. Hand optimization is, however, limited in its scope—typically scope of a routine. Hand optimization does not address redundancies that happen across several layers of abstractions in modular software. We believe these are all characteristics of future scientific codes for the Exascale era. Our work makes the following useful observations related to this type of code architectures: 1) as modular software implies "modular" contexts, flow-sensitive and context-sensitive analyses yield good results; and 2) as runtimes are written with portability in mind, holistic, dynamic analyses that examine applications and runtimes in conjunction yield good results.

3.8.3 Instrumenting NWChem

The important design concerns related to instrumenting the program execution for our analyses are portability and overhead.

We aim to provide portability of NWChem together with the analysis to other hardware. We also aim to provide a portable analysis that can be applied to other code infrastructures. Consequently, our design choice was to intercept the lowest-level RDMA APIs to identify all remote memory accesses. This design choice allowed our algorithm and workflow to remain independent of the application, run-time environment, and the hardware. We intercept RDMA calls by performing link-time wrapping [46] of the lowest-level communication libraries. Link-time wrapping allows us to divert all the necessary RDMA calls through our instrumentation layer reliably without having to make any changes to the large application code base. Library interposing via `LD_PRELOAD` [164] will work similarly for dynamically loaded libraries. Note that when callers and callees are defined in the same file, link-time approaches need to be augmented with source modifications. Other portable runtimes such as GASNet also present a single interface that resolves into system specific API calls. Note that portability is enabled by the fact that we are interested in the presence of these operations at runtime, rather than their exact semantics.

On the Cray hardware, the DMAPP layer provides a low-level system communication API. In any scientific application running on Cray's supercomputers, the higher-level communication abstractions eventually resolve to some DMAPP calls for RDMA operations. By link-time redefinition of DMAPP RDMA operations, we can intercept several configurations of NWChem such as its Global Arrays abstraction running on ARMCI, ComEx, or MPI. We have also audited the InfiniBand verbs API and believe our approach trivially ports to that system API.

To correctly elide synchronization, the analysis needs to detect accesses to local memory that bypass the standard RDMA calls for efficiency sake. We provide a plugin functionality that the developers can use to mark regions of code that perform such bypassing. As overhead and precision are a concern, the art is to identify the right level of granularity. With the cooperation of NWChem developers, we have audited the code, identified operations at

different levels of abstraction, and understood their side effects. We then manually inserted the instrumentation at only two places in the NWChem code to recognize all local accesses that have global visibility. These are the `comex_get_nbi` and `comex_put_nbi` calls, which resolve in either remote-access DMAPP calls or local memory accesses.

Another way by which local accesses can have global effects is by applications gaining a local pointer to the shared address space. Global Arrays have a well-defined interface for gaining a pointer access (`ga_access`) and releasing the access (`ga_release`) to the local shared regions. We intercept `ga_access` and `ga_release` with the same link-time wrapping. We count the number of `ga_access`es without the matching `ga_release`es at runtime. This accounting allows us to know if a code executed before a barrier might have updated the local shared memory without explicitly going through an RDMA call. We conservatively disable the automatic elision when there are outstanding `ga_access`es before a barrier.

Finally, for more flexibility, we expose APIs that an expert application developer can directly insert in the code base to explicitly enable or disable the automatic barrier elisions in the code regions of his choice. Several levels of global memory abstraction and the fact that "casting" sometimes occurs in an unprincipled manner based on code knowledge have complicated this process. Augmenting our guided analysis with a full-fledged data race detector would simplify this process. A principled approach to creating aliases between global and local pointers or using smart pointers can ease the task of a dynamic analysis framework in Partitioned Global Address Space programs.

3.8.4 Managing execution contexts

In our implementation, we identify contexts by unwinding the stack and computing a hash value based on the frame return address. We used `libunwind` [161] for unwinding and Google dense hash tables [76] for fast searching of contexts. We compiled the application by enabling frame pointers, and the overhead of unwinding was negligible in our experiments.

The automatic algorithm requires the calling contexts to be contextually aligned, i.e. the barriers do not need to be textually aligned [109] but their calling contexts always match the same way. At a barrier's calling context for a process, if the calling contexts of other processes participating in the same barrier do not remain stable, it will hinder the learning

process. Conservatively, we do not elide any barriers with contextual misalignment. To detect contextually misaligned barriers, we pass the context hash of each barrier in the reduction operation during the learning process. We drop contexts from eliding if we detect misalignment during learning. We note that the context hash can be different on different processes for dynamically loaded binaries. By canonicalizing the instruction pointer as a <load-module:instruction-offset> pair, we can obtain system-wide consistent context hashes. One can also use program debugging information to construct canonical context hashes.

3.8.5 Portability to other applications

Our framework has a core analysis component and a data race detection component. The analysis component consists of our stack unwinder, automatic and guided analyzers, and runtime elider. The analysis component can be used out of the box by any other application. The data race detection component has two subcomponents: a) application-agnostic RDMA instrumentation and b) application-specific instrumentation. The RDMA instrumentation is done once per specific system API (e.g., DMAPP). The RDMA instrumentation layer needs to be linked with the application in the last stage of the build for statically linked applications or should be preloaded for dynamically linked applications. The application-specific instrumentation needs to be neither too coarse-grained, nor too fine-grained and needs retargeting for each application. It took us about three months to get the application-specific instrumentation working correctly in NWChem. In general, application-specific instrumentation is needed when a pointer to a local shared datum escapes into a higher-level software layer, and the data is accessed without going through well-known communication APIs.

3.9 NWChem application insights

By examining barriers in conjunction with their dynamic behavior, we uncovered context-sensitive and context-insensitive redundant barriers. Most of the redundant barriers during NWChem execution are context sensitive.

Figure 3.7, presents the call graph of a routine inside the Hartree-Fock solver. The figure shows four lines of the application code written in Fortran (the leftmost box) along

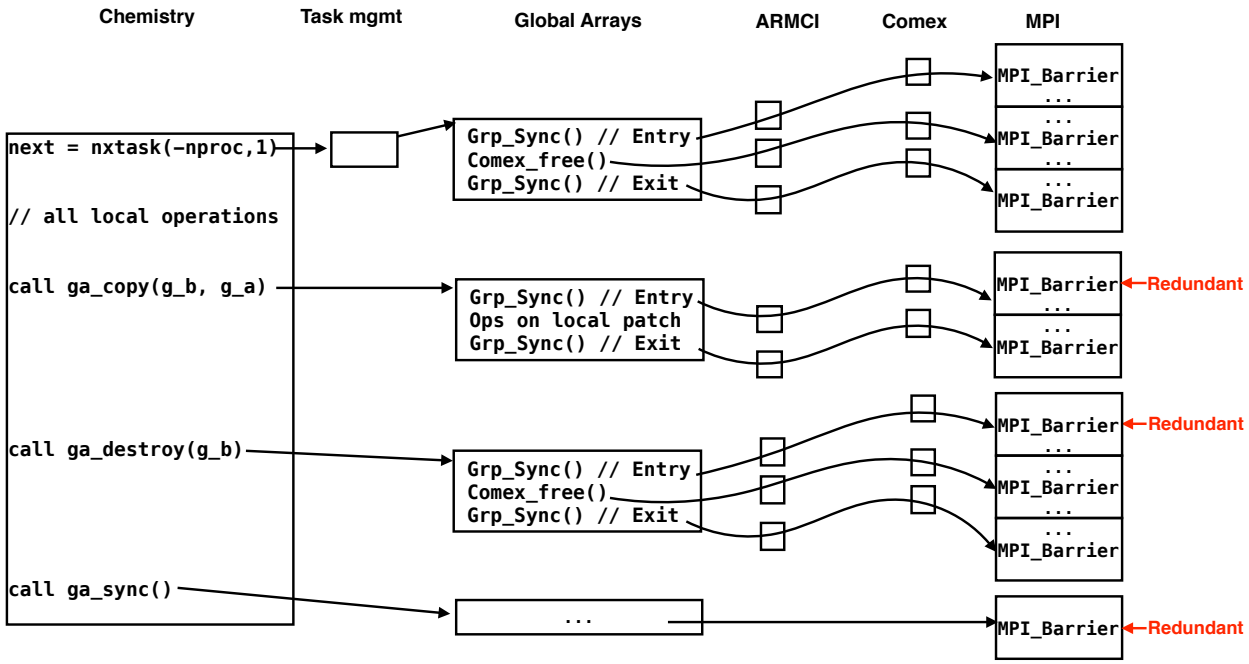


Figure 3.7 : Redundant barriers caused by software layering and API composition in NWChem. Synchronization is enforced on entry and exit of many API calls. Just four lines of the application code cause nine barriers at runtime. Three out of nine barriers (marked in red) are redundant since there would be no updates to shared data between two consecutive barriers.

with its call graph obtained by exploring several layers of the software stack as shown. The application code destroys an atomic task counter, copies the data from global memory to local memory, destroys the global memory, and performs barrier synchronization as the last step (unaware of the barriers enforced behind the scene). Synchronization is enforced on entry and exit of many API calls (middle boxes under Global Arrays). There are nine `MPI_Barrier` calls at runtime for just four lines of the application code. Three of nine barriers (marked in red) are redundant since there would be no updates to the shared data since the preceding barriers. In this case, the redundancy is context sensitive. Redundant barriers cannot be *textually removed* from the source code.

Figure 3.8 shows the call graph of the `pnga_add_patch` routine, where `pnga_destroy` is sometimes called in sequence based on the input arguments. Internally, `pnga_destroy` calls `MPI_Barrier` at entry and exit. Notice that the barrier enforced on entry to the second `pnga_destroy` is redundant if the first condition, `A_created`, is true. This is a flow-sensitive redundant barrier. Moreover, the programmer adds a `pnga_sync` call that causes another

redundant `MPI_Barrier`.

We also uncovered context-insensitive barriers. Figure 3.9 shows how inside `comex_malloc` and `comex_free`, a barrier operation is dominated by another collective operation, in this case, an `MPI_Allgather`, which implicitly has a barrier.

Figure 3.10 shows one of the top subcontexts found by our guided analysis. The redundant barrier is common to a group-operation function (`do_gop`). 8% of redundant barriers (11186 out of 138072) happen in this subcontext. 7% (553 out of the total 7959) of the unique redundant barrier contexts have this suffix in their calling contexts. On investigation, we found that in the `do_gop` routine, redundant barriers are intentionally introduced for portability, but they are not needed in most production system software configurations. These occur to quiesce the caller runtime upon any transition into a callee runtime, e.g., when transitioning from ComEx into an explicit independent MPI call inside the application. Our analysis can be configured to keep these barriers when needed.

The key insight is that redundancy is determined by clustering of calls at several levels of abstraction removed from the actual operation. In the NWChem case, this spans different programming languages and runtime implementations. Understanding the code by mere visual inspection is probably beyond most humans, and our analysis techniques provide useful insight to developers.

Most redundant barriers are context sensitive, but our subcontext-based classification indicates that only a few contexts contribute to a large fraction of the redundancy. In these cases, a synchronized/non-synchronized dual implementation is feasible at the application level.

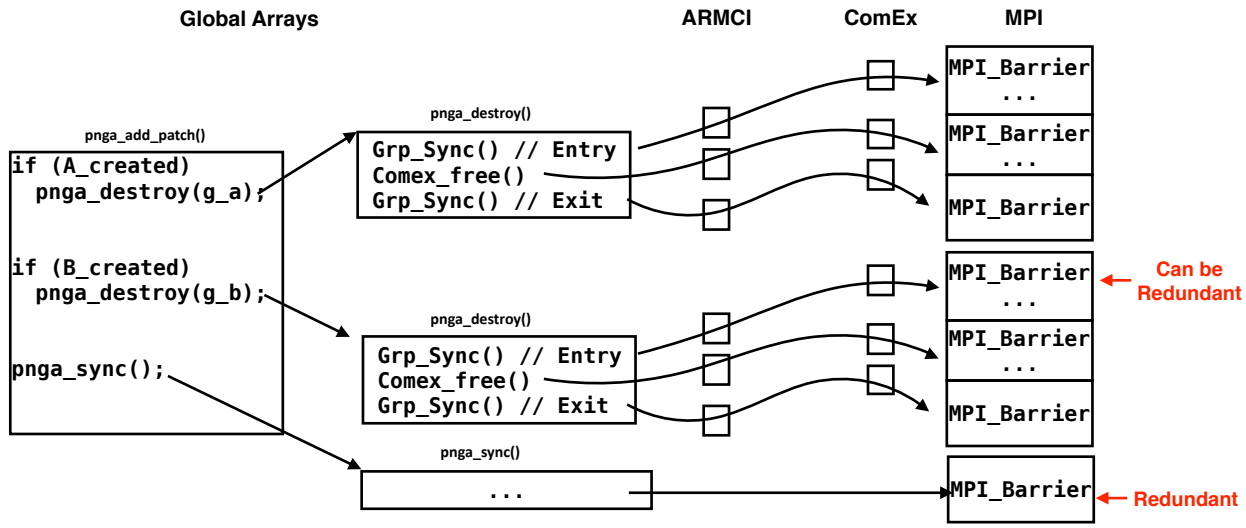


Figure 3.8 : Context- and flow- sensitive redundant barriers. Each pnga_destroy performs 3 barriers. If `A_created` is true, the barrier on entry to the second `pnga_destroy` is redundant. The last barrier is always redundant.

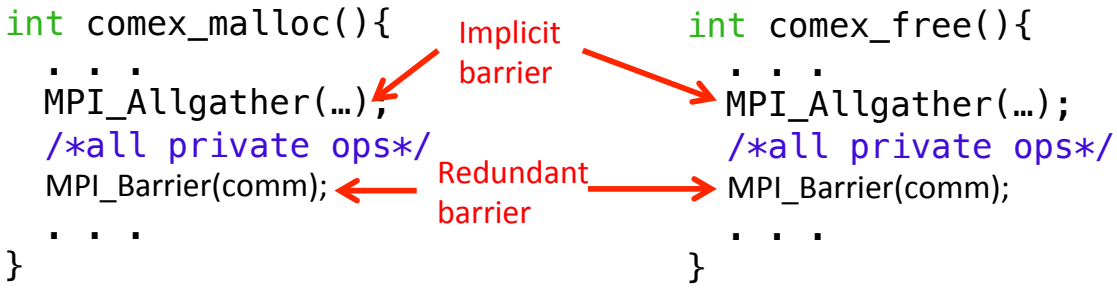


Figure 3.9 : Context-insensitive `MPI_Barrier` in ComEx. `MPI_Allreduce` has an implicit barrier and all subsequent operations are private.

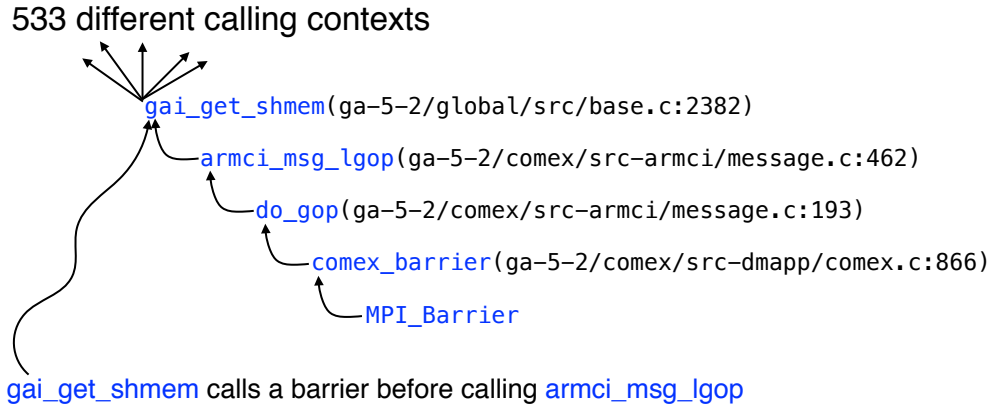


Figure 3.10 : Common subcontext of a top redundant barrier in NWChem. `MPI_Barrier` via the call chain starting at `armci_msg_lgop` are always preceded by another `MPI_Barrier`.

3.10 Performance evaluation

We evaluate performance on a Cray XC30 MPP installed at the National Energy Research Scientific Computing Center (NERSC). Each of its 5200 nodes contains two 12-core Ivy Bridge processors running at 2.4 GHz. Each processor has four DDR3-1866 memory controllers, which can sustain a stream bandwidth in excess of 50 GB/s. Every four nodes are connected to one Aries network interface chip. The Aries chips form a 3-rank dragonfly network [71]. Note that depending on the placement within the system, traffic can traverse either electrical or optical links. First, we present some basic findings on a microbenchmark. We then present our findings in NWChem production scientific runs. Finally, we present our findings related to the memory overhead of our tool.

3.10.1 Microbenchmarks

In Figure 3.11, we present results from a microbenchmark written to assess the runtime overhead of our implementation. The code performs 500,000 barrier operations. We vary the degree of barrier redundancy from 0% to 100%. We spread the barriers across 1,000 different calling contexts. We make our calling contexts 16 call stacks deep. To indicate that a barrier is necessary, the microbenchmark calls a dummy function before calling the barrier; this dummy function sets a flag in our instrumentation library. The performance is presented for a run using 9,600 MPI ranks. The figure indicates two things: first, the analysis adds a negligible runtime overhead per barrier operation, and second the elision can improve performance as the degree of barrier redundancy increases. At lower levels of barrier redundancy (10% and 20%), there is some variation in the running time from run to run leading to the zigzag pattern appearing in Figure 3.11. The pattern is not deterministic since the execution time varies due to other applications injecting their messages into the same shared network fabric. When the barrier redundancy increases, our algorithm performs fewer barriers, the execution becomes less communication bound, and the variation diminishes.

Overall, the performance improvements are determined by the scalability of barrier operations and the scalability of the analysis. Barrier latency grows with the number of cores. For example, we observe $30\mu s$ and $130\mu s$ at 2400 and 19200 processors, respectively. Overall,

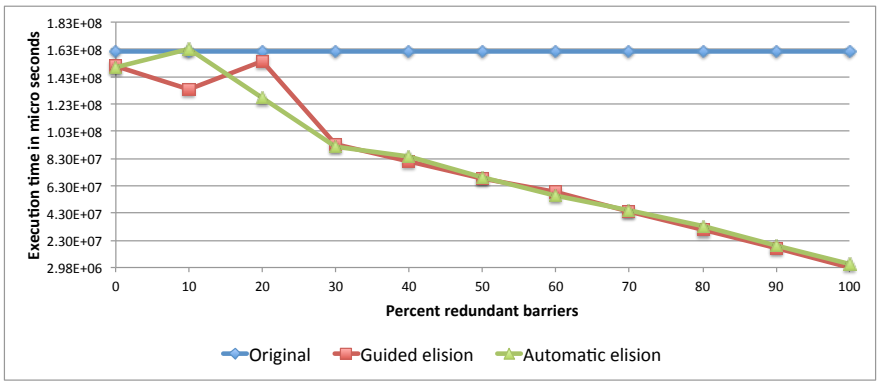


Figure 3.11 : Impact of fraction of redundant barriers on execution time for barrier elision microbenchmark. As more barriers become redundant, execution time reduces because of barrier elision.

the analysis overhead is independent of the number of cores but increases with the number of barrier contexts during execution and with the depth of the program stack. The guided analysis has a slightly lower overhead than the automatic analysis. For example, the overhead per barrier operation varies from 4.7μs to roughly 8μs when increasing stack depth from 4 to 64 and providing thousands of contexts. The maximum stack depth for any barrier during the NWChem runs is 21.

3.10.2 NWChem production runs

Table 3.5 presents the end-to-end performance improvements observed for two aforementioned science production runs of NWChem. As shown, both methods can uncover many redundant barriers, up to 45% and 63% for automatic and guided analyses, respectively. Elimination of these redundant barriers translates into application speedup up to 13.3% and 13.9% on 2048 cores, with automatic analysis and guided analysis respectively. Note that we report end-to-end speedup, which in NWChem includes a significant I/O portion.

Since the automatic algorithm learns behavior for a tunable number of repetitions of the same context, it may miss barriers that are redundant in many independent contexts. The algorithm learns only once, and it misses the cases where barriers become redundant later in the execution. To assess optimality, we compare the decisions of our automatic algorithm with the "true" barrier redundancy extracted using the coarse-grained data race information

Input	Number of cores	Time(s)	Total Barriers	Speed-up / Barriers elided guided	automatic	Unique Contexts
DCO	512	731	138072	0.7% / 63%	0.3% / 41.7%	7959
	1024	1084	138072	7.6% / 63%	0.2% / 41.4%	7959
	2048	1362	138072	13.9% / 63%	13.3% / 41.4%	7959
OCT	512	570	72188	3.4% / 63%	1.7% / 44.5%	4702
	1024	586	72188	6.6% / 63%	4.4% / 44.6%	4702
	2048	624	72188	6.0% / 63%	6.5% / 44.6%	4702

Table 3.5 : End-to-end performance results for the dichlorine oxide (DCO) and oxidized cytosine (OCT) simulations.

for an execution.

For the dichlorine oxide input, the code executes a total of 138,072 barriers in 7959 unique contexts. Only 45,614 barriers are required, according to our coarse-grained data race detection. The automatic algorithm learns for 31,750 barriers and elides 57,238 barriers in 2359 contexts, succeeding in deleting 41% of the redundant barriers. Remember that once a barrier is deemed essential it is never considered again for elision. During the speculation phase 5238 barriers episodes that were deemed essential become redundant during execution but are not skipped by our implementation. Similar trends are observable for other inputs. Overall, these results indicate that our techniques can skip a significant fraction of the redundant barriers. Extending the algorithms to re-learn redundancy is easy and may further improve performance.

Other performance improvements may be possible by specializing the learning process to exploit module information and dynamically tailoring the stabilization threshold. For example, for our inputs we set 10 as the learning threshold, whereas maximum threshold required was seven, and most of contexts stabilized by their 3^{rd} learning iteration.

3.10.3 Memory overhead

A key contributor to the memory overhead is the size of the hash table used to memorize the contexts. In both DCO and OCT scientific runs, the sizes of the hash tables were only 456KB per process. This size is insignificant compared to tens of Giga bytes of scientific data resident per process. Our instrumentation added 1MB binary code to an already 180MB NWChem's statically linked executable.

3.11 Discussion

Our online learning-based barrier elision approach is similar to common practices used in operating system job scheduling [212], branch prediction [152, 183, 241], prefetching [129, 231], and caching [18, 47, 52, 128, 244], among others. Learning-based approaches work well when applications have phases of behavior and are thus predictable. Of course, such techniques can easily be rendered wrong and driven to make worse decisions than they would have with no knowledge at all. One must use his or her judgment before employing such techniques in production.

The analyses exploit some inherent characteristics of the NWChem code base, which composes multiple iterative solvers written using SPMD parallelism. As the automatic method learns behavior, iterative algorithms present more optimization opportunity. Fortunately, this is the case with many scientific codes, which eschew direct solvers in favor of indirect iterative solvers. To make the overhead of classification palatable for the guided approaches it is desired that behavior learned at low concurrency applies to high concurrency. This is the case for most existing SPMD codes. For the few scientific codes that have been tuned to switch solvers based on concurrency, training at the appropriate concurrency or writing synthetic tests for solvers is required. Overall, we believe that this type of context-sensitive, dynamic analysis applies to many other scientific codes.

The combination of an ad-hoc, lightweight, data race detector with context-sensitive voting in synchronization operations (barriers) enables even more powerful synchronization optimizations. We are already considering extensions for reasoning about barriers in conjunction with other collectives, for transforming blocking collective calls into non-blocking calls, and for relaxing conservative communication synchronization operations such as fences. We believe these optimizations are useful for the Molecular Mechanics solvers in NWChem, whose scalability is limited by collective operations. Other code bases such as Cyclops Tensor Framework [216] will directly benefit from similar optimizations.

Mining the context information generated by our analyses, already provided insight into the code characteristics, which can be used for manual transformations. We believe there is an opportunity to refine the notions of context (e.g., to clusters of variables) and to extend the

classification methods to develop useful program understanding tools for large-scale codes.

Our approaches are not sound and use dynamic program analysis and speculation. Our technique is guaranteed to catch bad speculation at runtime. Due to the predictable behavior of the NWChem code, we never encountered a bad speculation; consequently, we did not employ checkpointing in our production runs. Like any other program transformation tool, developer confidence has to be gained through testing coverage for an arbitrary application.

Chapter 4

Tailoring Locks for NUMA Architectures

It is the accuracy and detail inherent in crafted goods that endows them with lasting value. It is the time and attention paid by the carpenter, the seamstress and the tailor that makes this detail possible.

Tim Jackson

Mutual exclusion is a technique for ensuring race freedom when accessing shared data in parallel programs. Despite hardware support for transactional memory [82, 87, 194, 234], locks remain as the most popular synchronization mechanism to achieve mutual exclusion. While non-blocking algorithms [119, 120, 127, 158] ensure atomicity of operations, they are difficult to implement for anything other than simple data structures.

A common cause of synchronization overheads in parallel programs is the overhead of inefficient locks. Despite being extensively explored [157], locks remain an active research area due to the changing architectural landscape. Many recent studies show that inefficient locks cause catastrophic performance losses under high contention [107, 125, 184]. In this chapter, we discuss the state-of-the-art in locks and offer our novel solutions tailored for the *shared-memory* NUMA architectures. The solutions offered in this chapter can be naturally extended to *distributed-memory* parallel architectures, although, we leave a concrete implementation and evaluation on such architectures as future work.

4.1 Motivation and overview

Over the last decade, the number of hardware threads in shared-memory systems has grown dramatically. Multiple hardware threads executing on a core share one or more levels of

private cache. Multiple cores share caches on the same die. Multi-socket systems share memory on a node, and multi-node shared-memory systems (e.g., SGI UV 1000 [208]) share memory across the entire system.

Modern architectures organize hardware threads into clusters known as NUMA (Non-Uniform Memory Access) domains. Each NUMA domain of threads experiences a spectrum of memory access latencies, depending upon the level of the memory hierarchy where the most recent copy of the data is resident (same core, same socket, or same node where the data is resident). Most importantly, in a cache coherent system, a datum gains proximity to a thread when the datum is accessed by the thread itself *or by other threads in the same NUMA domain*. Conversely, a datum loses proximity when a thread farther away in the NUMA hierarchy modifies the datum. The latency incurred by a thread when accessing a datum depends on the NUMA domain of the thread that accessed it last. Additional factors such as the proximity of the directory servicing the location and cache coherence protocol employed by the hardware for forwarding a cached copy of a location may also influence the access latency of a datum. Efficient use of deep NUMA systems requires exploiting locality. Once a memory location is cached in a NUMA domain, it is beneficial to reuse it multiple times. Ideally, the same thread or its NUMA peers would reuse the data. The next best alternative is access by threads in a NUMA domain nearby.

Figure 4.1 illustrates a typical 4-level NUMA system. Each pair of SMT threads (**S**imultaneous **M**ulti **T**hreading [227]—the most common style of hardware multithreading) sharing a core forms the first level of NUMA hierarchy. M different CPU cores sharing the same socket form the second level of NUMA hierarchy. K sockets on the same node form the third level of NUMA hierarchy. All nodes together form the fourth and last level of NUMA hierarchy.

Efficient locking mechanisms are critically important for high performance computers. Deep NUMA hierarchies pose a challenge for efficient mutual exclusion in multi-threaded programs. In the 1990s, Mellor-Crummey and Scott [157] designed a lock (MCS lock) implemented as a linked list of waiting threads. A few years later, Magnusson et al. introduced another lock design [146] for a queueing lock (often referred to as the CLH lock), where threads also spin on a unique flag. An advantage of queueing locks is that after a thread

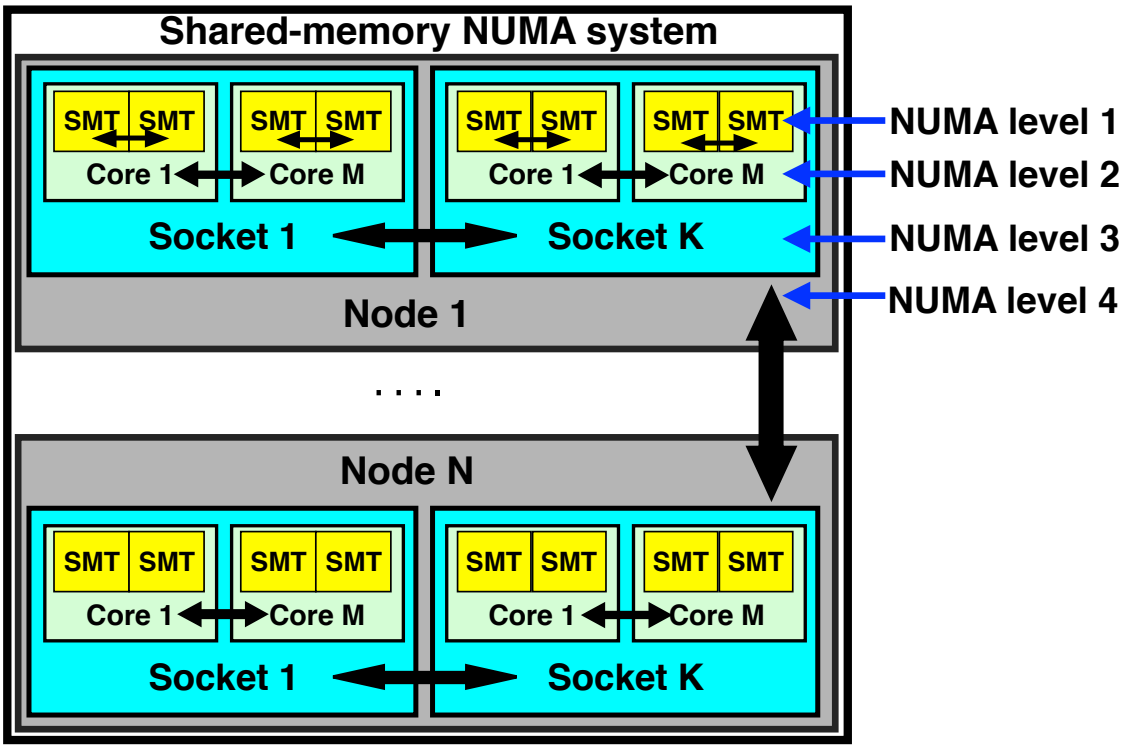


Figure 4.1 : Shared-memory NUMA system. A pair of SMT threads sharing a core forms the first level of NUMA hierarchy. M different CPU cores sharing the same socket form the second level of NUMA hierarchy. K sockets on the same node form the third level of NUMA hierarchy.

enqueues itself for a lock, it busy waits locally on a unique memory location, which avoids clogging the interconnect. For this reason, queuing locks scale well in the presence of contention. Queue-based locks such as MCS lock and CLH lock, however, were designed for a flat memory hierarchy; as a result, their throughput falls off on machines with NUMA architectures.

Lock-passing latency as the minimum time to hand over a lock from a thread holding the lock to a thread waiting for the lock. On deep NUMA hierarchies, a lock and data accessed in a critical section protected by the lock, ping-pong between NUMA domains, resulting in lower lock throughput. Figure 4.2 shows how the lock-passing latency grows on an SGI UV 1000 as the data becomes farther away from the requesting processor core.

Dice et al. [63] partially address this problem with "lock cohorting", which increases lock throughput by passing the lock among threads within the same NUMA domain before passing the lock to any thread in a remote NUMA domain. This design addresses only a two-

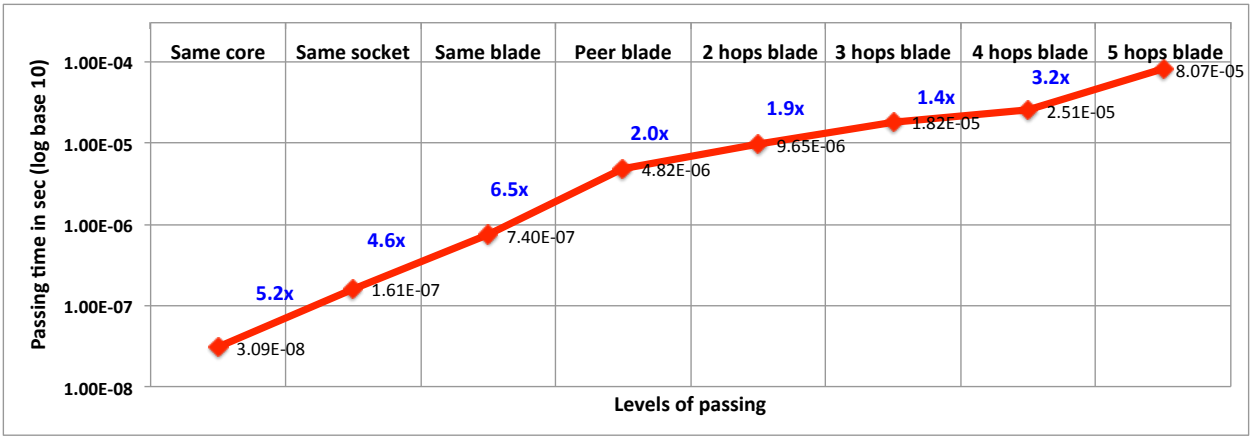


Figure 4.2 : MCS lock passing times on SGI UV 1000.

level NUMA hierarchy arising from multiple sockets on a node. For the system in Figure 4.1, cohort locks would only exploit locality among threads sharing the same socket, leaving other levels of locality unexploited. Deep NUMA hierarchies offer additional opportunities for exploiting locality among SMT threads sharing a cache, sockets within a node, among others.

In this chapter, we present hierarchical MCS locks (HMCS)—a full generalization of lock cohorting that takes advantage of each level of NUMA hierarchy. The HMCS lock is designed to provide an efficient mutual exclusion for highly contended critical sections in NUMA architectures. The HMCS lock is modeled as a composition of classical MCS locks at each level of NUMA hierarchy. By usually passing a lock to a waiting thread in the same or a nearby NUMA domain, the HMCS lock fully exploits locality on multilevel NUMA systems.

A limitation of multilevel locks such as the HMCS lock is their overhead under low contention, especially for uncontended acquisitions. To alleviate the overhead of HMCS lock under low contention, we design an *adaptive variant* of the HMCS lock that automatically adjusts to the contention levels in the system. The adaptive design has the advantage of *high throughput under high contention*, similar to the HMCS lock, and *low latency under low contention*, similar to the MCS lock.

4.2 Contributions

This work, part of which appeared in the Proceedings of the 20^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming [39], makes the following contributions.

1. It designs, implements, and evaluates a novel, fully-general, multilevel queuing lock suitable for mutual exclusion in deep NUMA systems,

2. It designs, implements, and evaluates analytical performance models for throughput and fairness of queuing locks on NUMA systems that eliminate empirical tuning,

3. It provides proofs for throughput and fairness superiority of multilevel queuing locks over state-of-the-art two-level locks under high contention,

4. It demonstrates up to 7.6× higher throughput over state-of-the-art two-level locks on a 128-thread IBM Power 755 and up to 72× higher throughput on a 4096-thread SGI UV 1000; it demonstrates 9.2× speedup compared to the original non-scaling K-means clustering code (55% improvement compared to the state-of-the-art two-level locks) on an IBM Power 755, and

5. It enhances the HMCS lock design to an adaptive lock that has desirable properties—high throughput under high contention and low latency under low contention.

4.3 Chapter roadmap

The rest of this chapter is organized as follows. Section 4.4 introduces the terminology and background necessary to understand the HMCS lock. Section 4.5 explains the HMCS algorithm. Section 4.6 discusses analytical performance models. Section 4.7 proves properties of HMCS locks. Section 4.8 presents an empirical evaluation of the HMCS locks. Section 4.9 enhances the HMCS lock to adapt dynamically to contention in a system. Section 4.10 augments the adaptive HMCS lock with the hardware transactional memory. Section 4.11 empirically evaluates the adaptive variants of the HMCS lock. Finally, Section 4.12 ends with some discussion.

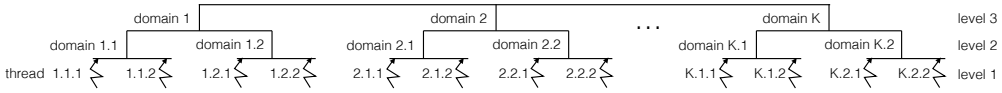


Figure 4.3 : Hierarchical tree of NUMA domains.

4.4 Terminology and background

Queuing lock algorithms form the basis for the current work. In this section, we define some terminology used in this chapter, followed by details about the MCS queuing lock and a two-level lock that is closest in its design to our lock.

Terminology: We refer to a system with N-levels of NUMA hierarchy as an *N-level system*. We refer to a NUMA domain as simply a *domain*. When the domain is important, we will refer to a *level-k domain*. Where necessary, we distinguish one thread (or domain) from another with dot-separated hierarchical numbering (Figure 4.3). A *level* in a hierarchy is counted starting from 1, from right to left in the dot-separated hierarchical numbering—(level n)$\cdots$(level 2).(level 1). Intuitively, levels are counted inside out starting from SMTs. For example, in Figure 4.1, thread 1.2.8.3 means the 3^{rd} SMT thread of the 8^{th} core of the 2^{nd} socket on the 1^{st} node. Two threads (or domains) are *peers* at level h, if they share a common prefix till level $h+1$. In Figure 4.3, threads **1.1**.1 and **1.1**.2 are peers at level 1, sharing the common prefix "1.1". Similarly, domains **1**.1 and **1**.2 are peers at level 2, sharing the common prefix "1". Note, however, that domains **1**.1 and **2**.1 are not level-2 peers since they do not share a common prefix at level 3. Two threads (or domains) that are *peers* at level h, *belong* to the same domains at levels greater than or equal to $h+1$. For example, threads **1.2**.1 and **1.2**.2 belong to the same level-2 (prefix "1.2") and level-3 (prefix "1") domains. Finally, we freely abbreviate "critical section" as *CS*.

MCS locks: The MCS [157] lock acquire protocol enqueues a record in the queue for a lock by: 1) swapping the queue's tail pointer with a pointer to its own record and 2) linking behind a predecessor (if any). If a thread has a predecessor, it spins locally on a flag in its record. Releasing an MCS lock involves setting a successor's flag or resetting the queue's tail pointer to null if no successor is present.

Two-level Cohort MCS locks: The Cohort MCS (C-MCS) lock by Dice et al. [63] for NUMA systems employs two levels of MCS locks, treating a system as a two-level NUMA hierarchy. (Dice et al. refer to this lock as C-MCS-MCS; we use the shorter C-MCS for convenience.) One MCS lock is local to each NUMA domain and another MCS lock is global—shared by all NUMA domains. Each thread trying to enter a critical section acquires the local lock first. The first thread to acquire the local lock in each domain proceeds to compete for the global lock while other threads in each domain spin wait for their local lock. A releasing thread grants the global lock to a local successor by releasing the local lock. A lock passes within the same domain for a *threshold* number of times, at which point the domain relinquishes the global lock to another domain. Dice et al. [63] also explore other lock cohorting variants beyond C-MCS that employ various locking protocols at each of the two levels, with the local and global locking protocols selected independently. We focus on MCS lock at each level of the HMCS lock and leave exploring different locks at different levels for future work.

4.5 HMCS lock algorithm

The HMCS lock extends the two-level cohorting strategy of Dice et al. [63] to a general N-level NUMA hierarchy. We abbreviate an N-level HMCS lock as HMCS$\langle N \rangle$. The HMCS lock employs MCS locks at multiple levels of the hierarchy to exploit locality at each level. An HMCS lock is an n-ary tree of MCS locks, shown in Figure 4.4. A critical section can be entered only when the acquiring thread holds all locks along a path from a leaf to the root of the tree. *Not all threads, however, explicitly acquire all locks along a path from leaf to root to enter the critical section.* A thread holding the lock, after finishing its critical section, passes the lock to a successor in its NUMA domain. As a result of this local passing, usually, threads can enter the critical section after only one MCS lock acquisition. After a *threshold* number of lock transfers within a NUMA domain, a thread passes the lock at the lowest enclosing ancestor NUMA domain where a thread from a peer domain is waiting. This chain of passing to a successor thread in the current or peer domain continues throughout the protocol. *Under high contention, threads acquiring the lock via lock passing, incur the cost of only a single leaf-level MCS lock acquisition.* The HMCS lock, like the C-MCS and HCLH

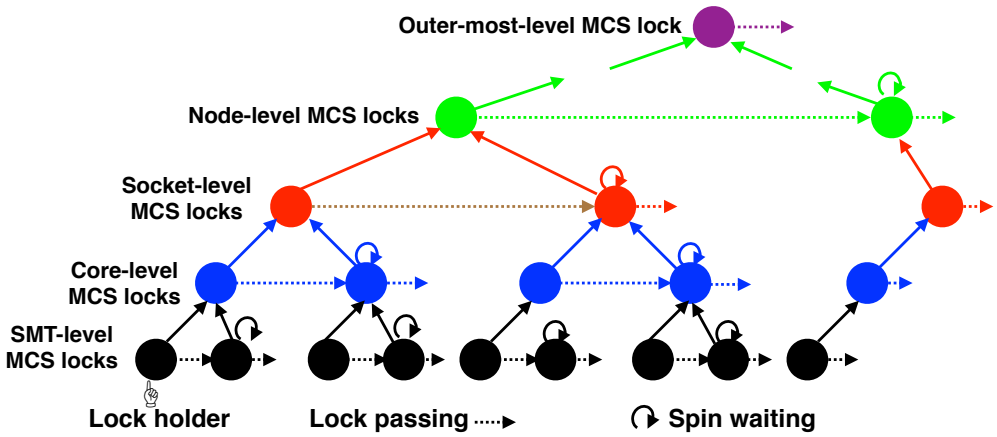


Figure 4.4 : Hierarchical MCS lock (HMCS). HMCS is composed of an n-ary tree of MCS locks, with one MCS lock at each level of the tree that mirrors the underlying structure of a NUMA architecture. Not all levels of the underlying NUMA architecture, however, need be mirrored. All hardware threads sharing the same CPU core use the leaf-level MCS lock designated for that core. All cores sharing the same socket share the MCS lock designated for that socket, and so on. The lock holder holds all locks from leaf a to the root. **Not all threads explicitly acquire all locks along the path from a leaf to the root but only a subset of locks starting from a leaf—typically just the leaf lock.** In steady state, the lock is passed from a predecessor to its local successor at the same level to take advantage of locality. There is a bound on local passing to ensure starvation freedom. When a successor is not present, or the threshold of local passing has reached, the global lock is relinquished to a thread waiting for the lock in the nearest ancestor NUMA domain.

locks, does not ensure system-wide FIFO ordering; however, it ensures FIFO ordering within requesting threads at the same level of the NUMA hierarchy.

Initial setup: Initialization of an HMCS lock involves creating a tree of MCS locks, with a lock for each domain at each level of the NUMA hierarchy. If exploiting locality at each level of the hierarchy is not profitable, one may create a shallower tree. For every non-leaf node of the tree, we allocate a lock record within the corresponding NUMA domain. These allocated records remain for the lifetime of the lock.

Key data structures: The HMCS algorithm employs two key data structures: `QNode` and `HNode`. The `QNode` is a modification of the original MCS lock's `QNode`, with its boolean status field replaced by a 64-bit integer. All `HNode`s representing a lock form a tree as shown in Figure 4.4. As shown in Figure 4.5, each `HNode` has the following fields:

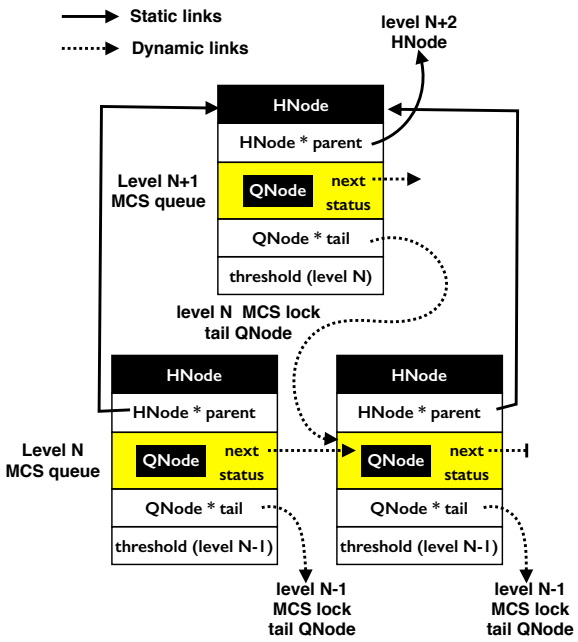


Figure 4.5 : HMCS lock key data structures.

- `lock`: the tail pointer of the MCS lock at this level,
- `parent`: a pointer to the parent `HNode` (if any),
- `QNode`: a node to enqueue when acquiring the parent `HNode`'s MCS lock. This `QNode` is shared by all threads in the subtree of the NUMA hierarchy represented by this `HNode`, and
- `threshold`: a bound on the number of times a lock can be passed to a successor at this level in the hierarchy. Based on the memory access latencies at different levels in a NUMA hierarchy, different levels in an HMCS tree can be configured with different passing thresholds to trade off throughput vs. fairness.

Acquire protocol: The HMCS lock algorithm is shown in Listing 4.1. The acquire protocol begins at the leaf-level (level-1) of the tree. Each thread arrives with its own `QNode` and a pointer to the lock—an `HNode` used by a group of peer threads at level-1 in the hierarchy. Each thread attempts to acquire the local level-1 MCS lock. The first thread to enqueue at each level-1 domain acquires the local MCS lock; other threads in the same domain spin-wait on the status field of their `QNode`. The first thread to acquire the level-i MCS lock in its NUMA domain proceed to compete with peers at the next level for the level-$(i+1)$ MCS

lock. This chain of acquisitions continues until a single thread acquires all MCS locks along the path to the root (inclusive), at which point the acquisition is complete, and the thread enters the critical section.

A thread (say `T`) that is not the first to arrive at a level (say K) spin waits on the `status` field of the enqueued `QNode` at level K (line 13 in Listing 4.1). Eventually, one of the following happens for a thread waiting at any level:

- (line 14 in Listing 4.1) `T`'s predecessor in the MCS lock at level K passes the lock to `T`, which completes the acquire protocol. `T` immediately enters the critical section without having to compete for the locks at levels greater than K.
- (line 14 in Listing 4.1) The quantum of lock passing exhausts at level K, in which case `T` proceeds to compete for the level-$(K+1)$ MCS lock (line 16, 17 Listing 4.1). Thread `T` also prepares for a full quantum of passing for the next round within the domain by resetting the `status` field of that domain's `QNode`.

Release protocol: The release protocol begins at the leaf of the tree. Each level in the tree is preconfigured with a bound on the number of times the lock can be passed in a domain at that level. This passing limit is recorded in the `threshold` field of every `HNode` designated for a domain at each level. When releasing the lock, if the number of local passes has not hit the threshold at the current level and there exists a waiting successor within that domain, then the releaser (`R`) passes the lock to its waiting peer within the domain. Otherwise, `R` relinquishes the lock at the nearest ancestor node along the path to the root where 1) a peer is waiting, and 2) the passing threshold has not been exceeded at that level.

While the classical MCS lock uses a boolean flag to pass a lock to its successor, the HMCS lock uses a 64-bit integer (the `status` field in each `QNode`) to encode the current local pass count. When passing the lock to a successor at the same level of the HMCS tree, the releaser `R` 1) reads the 64-bit value `V` present in the `status` field of its `QNode` and 2) writes `V+1` into the `status` field of its successor if and only if `V` is less than the passing `threshold` at that level. Writing `V+1` into the successor's `status` field serves both to pass the lock to the successor as well as to convey the local pass count from `R` to its successor. This technique eliminates the need for a shared counter for bookkeeping the pass counting within a domain.

```
1  enum {COHORT_START=0x1, ACQUIRE_PARENT=UINT64_MAX-1, WAIT=UINT64_MAX};
2  enum {UNLOCKED=0x0, LOCKED=0x1};
3
4  template<int depth> struct HMCSLock {
5    inline void AcquireReal(HNode *L, QNode *I) {
6      // Prepare the node for use.
7      I->status = WAIT; I->next = NULL;
8        ------------------- release fence -------------------
9      QNode * pred = (QNode *) SWAP(&(L->lock), I);
10     if (pred) {
11       pred->next = I;{
12       uint64_t curStatus;
13 ▶      while( (curStatus=I->status) == WAIT); // spin
14 ▶      if(curStatus < ACQUIRE_PARENT)  return; // Acquired, enter CS
15     }
16     I->status = COHORT_START;
17     HMCSLock<depth-1>::AcquireReal(L->parent, &(L->node));
18   }
19
20   inline void Acquire(HNode *L, QNode *I) {
21     AcquireReal(L, I);
22 ▶     ------------------- acquire fence -------------------
23   }
24
25   inline  void ReleaseReal(HNode *L, QNode *I) {
26 ▶   uint64_t curCount = I->status;
27     // Lower level releases
28 ▶   if (curCount == L->GetThreshold()) {
29       // reached threshold, release to next level
30 ▶     HMCSLock<depth-1>::ReleaseReal(L->parent, &(L->node));
31       ------------------- release fence -------------------
32       // Ask successor to acquire next level lock
33       ReleaseHelper(L, I, ACQUIRE_PARENT);
34       return; // Released
35     }
36     // Not reached threshold
37 ▶   QNode * succ = I->next;
38 ▶   if (succ) {  // Pass within cohorts
39 ▶     succ->status = curCount + 1;
40       return;
41     }
42     // else: No known successor, release to parent
43    HMCSLock<depth-1>::ReleaseReal(L->parent, &(L->node));
44       ------------------- release fence -------------------
45     // Ask successor to acquire next level lock
46     ReleaseHelper(L,I, ACQUIRE_PARENT);
47   }
48
49   inline void Release(HNode *L, QNode *I) {
50 ▶     ------------------- release fence -------------------
51     ReleaseReal(L, I);
52   }
53 };
54
55 void ReleaseHelper(HNode *L, QNode *I, uint64_t val){
56 ▶   QNode * succ = I->next;
57 ▶   if (succ) {
58 ▶     succ->status = val; // pass lock
59    } else {
60       if (CAS(&(L->lock), I, NULL)) return;
61       while( (succ=I->next) == NULL);  // spin
62       succ->status = val; // pass lock
63    }
64 }
65
66 template <> struct HMCSLock<1> {
67   inline void AcquireReal(HNode *L, QNode *I) {
68     // Prepare the node for use.
69     I->status = LOCKED; I->next = NULL;
70     QNode * pred = (QNode *) SWAP(&(L->lock), I);
71     if (!pred) {
72       I->status = UNLOCKED;
73     } else {
74       pred->next = I;
75 ▶       while(I->status == LOCKED); // spin
76     }
77   }
78
79   inline void Acquire(HNode *L, QNode *I) {
80     AcquireReal(L, I);
81 ▶     ------------------- acquire fence -------------------
82   }
83
84   inline void ReleaseReal(HNode *L, QNode *I) {
85     ReleaseHelper(L, I, UNLOCKED);
86   }
87
88   inline void Release(HNode *L, QNode *I) {
89 ▶     ------------------- release fence -------------------
90     ReleaseReal(L, I);
91   }
92 };
```

Listing 4.1: The Hierarchical MCS lock algorithm.

When releaser R encounters a pass count V that has exceeded the passing threshold inside a domain (represented by an HNode L), other than the tree root, R recursively performs the release protocol (line 30 in Listing 4.1) at L->parent before signaling its successor S. If S is non-null, R signals S by setting S's status to ACQUIRE_PARENT, which indicates that S must compete for the MCS lock at its parent level (L->parent). If R were to signal S before releasing at L->parent, there would be a data race between R and S for the QNode they both use (L->node) to interact with the MCS lock at the parent level.

Attempting to pass a lock to a thread in the same NUMA domain before relinquishing the lock to the next level enhances reuse of shared data. Limiting the number of local passes ensures starvation freedom. If the passing threshold is exceeded but there is a successor at the current level, whereas no threads in other domains are waiting, then we retain the lock within the domain for another round. This optimization is not shown in Listing 4.1.

Relaxed memory models: Listing 4.1 shows the fences necessary for the HMCS lock on systems with processors that use weak ordering.

1. The release fence on line 8 ensures that the status and next fields of the QNode are initialized and visible for the predecessor.

2. The acquire fence on line 22 ensures that the accesses inside the critical section are not reordered with instructions in the acquire protocol.

3. The release fences on line 31 and line 44 ensure that the release of a parent lock and updates to a shared QNode are visible to a successor sharing the same parent, before the successor node is signaled.

4. The release fence on line 50 ensures that the accesses inside a critical section are not reordered with respect to the writes after the critical section.

5. The acquire fence on line 81 ensures that the accesses inside the critical section are not reordered with instructions in the acquire protocol.

6. The release fence on line 89 ensures that the accesses inside a critical section are not reordered with respect to writes after the critical section.

4.5.1 Correctness

In this section, we provide correctness proof of the HMCS lock. An HMCS lock does not ensure the FIFO property. We need to show the following two aspects for the correctness of an HMCS$\langle N \rangle$ lock, for any value of N:

1. An HMCS$\langle N \rangle$ lock ensures mutual exclusion.
2. An HMCS$\langle N \rangle$ lock ensures bounded waiting [212].

4.5.1.1 Proof for the mutual exclusion of HMCS$\langle N \rangle$

Axiom 4.1 *MCS lock ensures mutual exclusion [108, 155].*

Since HMCS$\langle 1 \rangle$ is same as MCS, it trivially follows from Axiom 4.1 that HMCS$\langle 1 \rangle$ ensures mutual exclusion.

We follow an inductive argument to prove the mutual exclusion property of an HMCS$\langle N \rangle$ lock.

Base case ($N = 2$): First, we show that when $N = 2$, HMCS$\langle 2 \rangle$ ensures mutual exclusion. *Proof by contradiction:* Assume HMCS$\langle 2 \rangle$ does not ensure mutual exclusion. Hence, HMCS$\langle 2 \rangle$ must allow two threads to be simultaneously present in the CS. If so, these two threads (say T_1 and T_2), cannot be from two peer threads (different domains) at level 2. This because, either T_1 itself acquired the level-2 MCS lock or some *level-1 peer* of T_1 acquired the level-2 MCS lock prior to granting the level-1 MCS lock to T_1. In either case, the level-2 lock is held by T_1's innermost enclosing domain and is not released yet. If T_1 and T_2 are peers at level 2, then they compete for the same level-2 MCS lock; and by the mutual exclusion property of the level-2 MCS lock, no two threads can simultaneously acquire the same MCS lock. Hence, no two level-2 peers can simultaneously be present in a CS. Note that the same argument holds if T_2 was the level-2 lock holder.

In a 2-level system, two threads are either level-1 peers or level-2 peers. If T_1 and T_2 cannot be level-2 peers, they must be level-1 peers, belonging to the same inner domain. Two level-1 peers compete for the same level-1 MCS lock. By the mutual exclusion property of the level-1 MCS lock, T_1 and T_2 cannot acquire the same level-1 MCS lock simultaneously. Hence, T_1 and T_2 cannot be level-1 peers either.

T_1 and T_2 cannot simultaneously be in the CS whether they are level-2 peers (belong to different domains) or level-1 peers (belong to same domain), which contradicts of our assumption. Hence, two threads cannot simultaneously be in a CS. Thus, HMCS$\langle 2 \rangle$ ensures mutual exclusion. This proof resembles the one in [64].

Inductive step: Since HMCS$\langle 2 \rangle$ ensures mutual exclusion, assume that HMCS$\langle M \rangle$, where $M > 2$, ensures mutual exclusion.

We now need to show that HMCS$\langle M + 1 \rangle$ ensures mutual exclusion. An HMCS$\langle M + 1 \rangle$ lock is composed of several HMCS$\langle M \rangle$ local locks and a root-level MCS lock.

Proof by contradiction: Assume an HMCS$\langle M + 1 \rangle$ allows two threads to be simultaneously present in a CS. If so, these two threads, T_1 and T_2, cannot be peers (belong to different domains) at level M+1. This is because, if a thread T_1 is in the CS, then either T_1 itself acquired the level-(M+1) MCS lock or another thread that is a peer of T_1 at level less than or equal to M (and hence sharing the same HMCS$\langle M \rangle$ lock as T_1) released its HMCS$\langle M \rangle$ or a lower level lock allowing T_1 to enter the CS. In either case, the level-(M+1) MCS lock is held by a thread sharing the same domain as T_1 at level M+1, and the level-(M+1) MCS lock is not released yet. If T_1 and T_2 are peers at level M+1, then they compete for the same level-(M+1) MCS lock; but by the mutual exclusion property of the level-(M+1) MCS lock, no two threads can simultaneously acquire the same MCS lock. Hence, no two level-(M+1) peers can simultaneously be present in a CS.

In an (M+1)-level system, two threads are peers at some level less than or equal to M+1. If T_1 and T_2 cannot be level-(M+1) peers, then they must be peers at a level less than or equal to level-M, sharing the same HMCS$\langle M \rangle$ lock. By the mutual exclusion property of the HMCS$\langle M \rangle$ lock (inductive step), T_1 and T_2 cannot simultaneously acquire the same HMCS$\langle M \rangle$ lock. Hence, T_1 and T_2 cannot be peers at any level less than or equal to M either.

T_1 and T_2 cannot simultaneously be in the CS whether they are peers at level M+1 (belong to different domains) or peers at a level less than M+1 (belong to the same domains), which contradicts of our assumption. Hence, two threads cannot simultaneously be in a CS in an HMCS$\langle M + 1 \rangle$ lock. Hence, HMCS$\langle M + 1 \rangle$ ensures mutual exclusion.

By induction, HMCS$\langle N \rangle$ ensures mutual exclusion.

4.5.1.2 Proof for the bounded waiting in HMCS$\langle N \rangle$

Axiom 4.2 *MCS lock ensures FIFO ordering of threads requesting to enter a critical section once they have swapped the tail pointer. This also means the MCS lock ensures a bounded wait for the threads requesting to enter a CS [108, 155].*

Since HMCS$\langle 1 \rangle$ is same as MCS, it trivially follows from Axiom 4.2 that HMCS$\langle 1 \rangle$ ensures FIFO property, which is stronger guarantee than bounded waiting. It follows from the FIFO property of the MCS lock that the bound on waiting of an MCS lock with n contenders is n.

Let n_i be the number of contenders at level i. Let h_i be the passing threshold at level i.

For HMCS$\langle 2 \rangle$, when $h_1 \geq n_1$, the longest waiting happens for the last thread (say t_l) enqueued in level-1 MCS lock of the last domain (say D_l) enqueued at level-2 MCS lock. The following three quantities play role in determining the bound on waiting:

1. If t_l has enqueued in D_l but the head of D_l (possibly t_l itself) is yet to enqueue at level-2, there can be a small, finite, but non-deterministic number (say δ) of lock acquisitions elsewhere in the system during such interval.

2. Each domain hands out a maximum of h_1 locks in a FIFO order to its level-1 MCS contenders before relinquishing the lock at level-2. Level-2 MCS lock also obeys the FIFO property. By the time the lock is passed to domain D_l, there could be as many as $(n_2 - 1)h_1$ lock acquisitions in the entire system.

3. In the last domain, D_l, there can a maximum of n_1 lock acquisitions before the longest waiting thread t_l can enter its critical section.

Summing up the three quantities, the upper bound $\mathcal{B}$ on waiting is:

$$\begin{aligned} \mathcal{B}_{hmcs\langle 2 \rangle} &= (n_2 - 1)h_1 + n_1 + \delta \\ &\leq (n_2 - 1)h_1 + h_1 + \delta \\ &\leq n_2 h_1 + \delta \end{aligned} \tag{4.1}$$

For HMCS$\langle 2 \rangle$, when $h_1 < n_1$, each domain gets lock acquisitions in h_1 quantum. The longest waiting thread (t_l) will be the last enqueued thread in the last domain D_l enqueued

at level 2. Because of the FIFO property of the MCS lock used at level 1, all $n_1 - 1$ threads that are ahead of t_l in D_l should be served before t_l can enter the CS. This means, t_l is assured of the CS entry only after D_l has obtained the level-2 MCS lock $\lceil \frac{n_1}{h_1} \rceil$ times. This means, there will be $\lceil \frac{n_1}{h_1} \rceil h_1$ lock acquisitions in D_l before t_l finishes. Each time D_l wants to acquire the level-2 lock, it may be the last one to do so at level-2, forcing it to wait for $n_2 - 1$ level-2 lock acquisitions ahead of it—a total of $\lceil \frac{n_1}{h_1} \rceil (n_2 - 1)$ level-2 MCS lock acquisition. Each of those $\lceil \frac{n_1}{h_1} \rceil (n_2 - 1)$ level-2 acquisitions will handout at most h_1 locks in other domains for a total of $\lceil \frac{n_1}{h_1} \rceil (n_2 - 1)h_1$ lock acquisitions. As before, let δ be the total, finite, but non-deterministic number of lock acquisitions elsewhere in the system between the time threads at the head of D_l on $\lceil \frac{n_1}{h_1} \rceil$ different occasions enqueue at the level-2 MCS lock. Summing up the three quantities, the upper bound $\mathcal{B}$ on waiting is:

$$\begin{aligned} \mathcal{B}_{hmcs\langle 2 \rangle} &= \left\lceil \frac{n_1}{h_1} \right\rceil h_1 + \left\lceil \frac{n_1}{h_1} \right\rceil (n_2 - 1)h_1 + \delta \\ &= \left(\left\lceil \frac{n_1}{h_1} \right\rceil n_2 \right) h_1 + \delta \end{aligned} \tag{4.2}$$

When $h_1 \geq n_1$, $\left\lceil \frac{n_1}{h_1} \right\rceil$ becomes 1 in Eqn 4.2 and hence equals Eqn 4.1. Thus, we see that in either case Eqn 4.2, which is a finite value, is the bound on waiting in HMCS$\langle 2 \rangle$.

We can easily extend Eqn 4.2 to HMCS$\langle 3 \rangle$. Each time a thread rises to acquire the level-3 lock, it might be at the end of the queue, waiting $n_3 - 1$ level-3 acquisitions ahead of it. Each time a level-3 lock is acquired by any domain, there can be $h_1 h_2$ additional lock acquisitions within such domain. The parenthesized term in Eqn 4.2 will be served in quanta of h_2 in an HMCS$\langle 3 \rangle$ lock. Hence, the bound on waiting in an HMCS$\langle 3 \rangle$ is:

$$\mathcal{B}_{hmcs\langle 3 \rangle} = \left(\left\lceil \left\lceil \frac{n_1}{h_1} \right\rceil \frac{n_2}{h_2} \right\rceil n_3 \right) h_1 h_2 + \delta$$

It can be shown that the bound on waiting in an HMCS$\langle N \rangle$ is:

$$\mathcal{B}_{hmcs\langle N \rangle} = \Big(\psi_{N-1} n_N\Big) \prod_{i=1}^{N-1} h_i + \delta \tag{4.3}$$

$$\text{where, } \psi_i = \left\lceil \left\lceil \left\lceil \frac{n_1}{h_1} \right\rceil \frac{n_2}{h_2} \right\rceil ... \frac{n_i}{h_i} \right\rceil$$

Thus, we see that Eqn 4.3, which is a finite value, ensures bounded waiting in HMCS$\langle N \rangle$.

4.5.2 Discussion

Latency vs. throughput: Clearly, the cost of occasionally manipulating a sequence of locks along a path in a tree is more expensive than working with a single lock. For highly contended locks, however, this cost is outweighed by the benefits of increased locality that result from sharing within a NUMA domain. The HMCS design improves throughput at the expense of latency, and it is *best suited for highly contended locks.* To reduce latency, a client can use a shallower HMCS tree though doing so would reduce the lock's peak potential throughput. With the template-based design, we can instantiate the HMCS lock as a classical MCS lock (HMCS$\langle 1 \rangle$), a 2-level cohort lock (HMCS$\langle 2 \rangle$), or use a larger template constant for a deeper hierarchy. Although, the code in Listing 4.1 uses compile-time instantiation, one could implement HMCS lock using runtime recursion instead of template meta programming.

Memory management: The HMCS lock needs no explicit memory management after initialization. In contrast, Dice et al.'s cohort MCS lock requires maintaining a pool of free nodes. For that reason, HMCS lock's memory management is superior.

Cache policy: Performance benefits of the HMCS lock can vary based on the caching policies such as inclusive vs. exclusive. By passing the lock to a nearby thread more often, the HMCS lock benefits from data locality *irrespective* of the caching policy.

Thread binding: The code in Listing 4.1 is agnostic to operating-system (OS) thread to hardware (HW) thread bindings. The thread calling the HMCS lock's `Acquire` and `Release` routines is responsible for using the correct `L`—the pointer to its leaf-level `HNode`. If the OS

thread is bound to a HW thread belonging to the same innermost NUMA domain, then the program can set the value of `L` once during lock initialization and use it for the duration of the program.

To address thread migration across NUMA domains, we replace the `L` argument in the `Acquire` and `Release` routines to point to an array of pointers to leaf-level `HNode`s. The `Acquire` *indexes* into this array to obtain the pointer to its `HNode`. The `Acquire` will also remember the inferred value of the `index`, which the matching `Release` will use. The `index` itself will be a cached, thread-local value derived by querying the CPU that the thread is currently running on. We register a signal handler with Linux perf events for task migration. On task migration, the handler invalidates the migrating thread's cached CPU `index`. The first acquire following a migration, incurs an additional cost of updating the `index` inferred either via an architecture-specific instructions such as `RDTSCP` or via the `getcpu` system call.

4.6 Performance metrics

Two key governing design criteria for any locks are their *throughput* and *fairness*. The HMCS and C-MCS locks are designed to deliver high throughput under heavy contention. For this reason, we evaluate throughput and fairness of these locks in the fully contended case. We do not have access to the original implementation of the C-MCS lock. We use HMCS$\langle 2 \rangle$ as a proxy for the C-MCS lock, albeit HMCS$\langle 2 \rangle$ is superior due to no memory management. In the rest of this chapter, a reference to C-MCS simply means HMCS$\langle 2 \rangle$.

One can evaluate locks under full contention in various ways. Mellor-Crummey and Scott [157] suggest continuous lock acquisitions and releases with an empty critical section to assess the *pure overhead* of locks. Dice et al. suggest adding a delay outside the critical section and accessing a few cache lines of shared data inside the critical section. Each evaluation method has its own advantages and disadvantages. Dice et al.'s technique of accessing sharing data can give a false impression of reduced lock overhead since the shared data is also passed between NUMA domains. The larger the shared data accessed in the critical section, the lower the observed overhead of the lock. Further, there is never an appropriate delay to choose for the time spent outside a critical section. On the other hand, Mellor-Crummey and Scott's tight loop ignores the actual gain possible on a real program,

which would certainly access some shared data inside critical section; passing shared data within NUMA domains has nontrivial benefits. Finally, if the data accessed inside a critical section is so large that it evicts critical lock data structures, both Dice and Mellor-Crummey's techniques are moot. Being aware of these differences and deficiencies, we adopt Mellor-Crummey and Scott's tight loop technique to quantify the pure lock overhead without any external interference from the program.

Let n_i be the number of peers in domain i. The lowest possible domain is the L1 cache of a core, so n_1 is the number of hardware threads per core. Analogously, n_2 is the number of cores per chip, n_3 is the number of chips / socket, etc. Given this definition, it is clear that the total number of threads in the system is $\prod_{i=1}^{N} n_i$.

4.6.1 Throughput

Definition 4.1 (Lock throughput) *Lock throughput, $\mathcal{T}_k$, of a lock k, is the average number of lock acquisitions by k per unit time.*

Definition 4.2 (Critical path) *The critical path is the longest path from beginning to end that has no wait states, where the edge weights correspond to operation latencies.*[1]

Naturally, in Mellor-Crummey and Scott's experimental setup, all statements in the critical section are on the critical path. The time to execute the statements in the critical section is a property of the program. For assessing lock overhead, however, we assume an empty critical section. All statements that are executed from the time the lock is granted to the time the lock is passed to the next waiting thread contribute to the critical path. Not all statements in the lock-acquire and lock-release protocols contribute to the critical path. Under high contention, a large number of threads will be waiting to acquire the lock. The thread releasing the lock will have a successor linked behind it at each level in the hierarchy. Each waiting thread will be spin-waiting at line 13 in Listing 4.1 in the acquire protocol. We have marked all statements that *may* appear on the critical path with a "▶" symbol in Listing 4.1.

[1]Spin waiting corresponds to the wait states.

The following statements in the acquire protocol appear on the critical path: 1) the last trip through the spin-wait loop after the lock is granted, and 2) the check to ensure the passing threshold was not exceeded.

The following statements in the release protocol appear on the critical path: 1) loading of the status field, 2) checking if the passing threshold has reached, 3) relinquishing the global lock to the parent level when the passing threshold is reached, and 4) checking for the presence of successor and subsequent global lock passing to the successor when the passing threshold has not reached.

If the lock is relinquished to the parent level, the sequence of statements executed until the global lock is granted to another domain contributes to the critical path. Subsequent signaling of successors at lower levels is *not* on the critical path. In a fully contended system, there is always a successor waiting to acquire the lock; hence, the case of not having a successor (line 57 in Listing 4.1) is not on the critical path.

Our implementation takes advantage of C++ template's value specialization to unroll recursion. Template specialization allows the deepest level of recursion, which manipulates the root lock, to be implemented with lower overhead. When the acquisition involves several layers of recursion, tail recursion reduces the critical path length since the lock acquiring thread can enter the critical section with a single return statement from the last level of recursion. Furthermore, we engineered the code to reduce branches on the critical path (not shown in Listing 4.1.)

A variant of the C-MCS lock: While Dice et al.'s C-MCS lock formed cohorts within threads on the same socket it ignored the other levels of NUMA hierarchy inside a socket. We propose a variant of C-MCS lock where cohorts are formed only at the inner level (e.g., among SMT threads). All other levels of the NUMA hierarchy are ignored. We call Dice et al.'s original C-MCS lock an *outer* cohorting lock (C-MCS$_{\text{out}}$). We call the aforementioned variant of the C-MCS lock as an *inner* cohorting lock (C-MCS$_{\text{in}}$).

For simplicity, in Section 4.6.1.1- 4.6.1.3 we assume a 3-level system and build analytical models of throughput for C-MCS$_{\text{in}}$, C-MCS$_{\text{out}}$ and HMCS$\langle 3 \rangle$ locks. We provide throughput models of the HMCS lock for any N-level system at the end of Section 4.6.1.3.

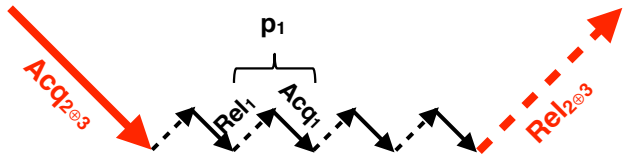


Figure 4.6 : Lock passing in the C-MCS$_{in}$ lock.

4.6.1.1 Throughput of the C-MCS$_{in}$ lock

In a C-MCS$_{in}$ lock, threads at level-1 of the NUMA hierarchy (e.g., SMT threads) form cohorts. On reaching the passing threshold, however, a C-MCS$_{in}$ lock may relinquish the lock either to a level-2 NUMA peer (e.g., a thread on the same socket) or to a level-3 NUMA peer (e.g., a thread on another socket). Consider a chain of successive lock passing as shown in Figure 4.6. The first thread in the cohort acquires the lock either from a level-2 or level-3 peer in the NUMA hierarchy. Let Acq_2 and Acq_3 be the critical path lengths if the lock is acquired from a level-2 or level-3 peer respectively. Since the level-3 memory access latency is larger than level-2 memory access latency, $Acq_3 > Acq_2$. The expected critical path length to acquire the lock either from a level-2 or level-3 NUMA domain peer is given by:

$$Acq_{2\oplus 3} = Pr[Acquisition\ from\ level\ 2]\ Acq_2 + (1 - Pr[Acquisition\ from\ level\ 2])\ Acq_3 \tag{4.4}$$

There are $(n_2 n_3 - 1)$ other domains in the system, of which $(n_2 - 1)$ are peers at level 2. Hence,

$$Pr[Acquisition\ from\ level\ 2] = \frac{n_2 - 1}{n_2 n_3 - 1} \tag{4.5}$$

$$\implies Acq_{2\oplus 3} = \frac{n_2 - 1}{n_2 n_3 - 1} Acq_2 + \frac{n_2 n_3 - n_2}{n_2 n_3 - 1} Acq_3 \tag{4.6}$$

Let Rel_1 and Acq_1 be the critical paths in the release and acquire protocols when releasing and acquiring at level-1 respectively. We represent $Rel_1 + Acq_1 = p_1$ as the *lock passing* time at level 1. If c_{in} is the passing threshold for the lock, there will be $(c_{in} - 1)$ locks passing at level 1, each adding p_1 to the critical path length. At the end, the lock will be released to a thread belonging to either level-2 or level-3 domains and the expected cost of releasing $(Rel_{2\oplus 3})$ is defined similar to $Acq_{2\oplus 3}$. We represent $Acq_{2\oplus 3} + Rel_{2\oplus 3}$ as $p_{2\oplus 3}$, where p_2 and

p_3 are the passing times at level 2 and 3 respectively. The expected passing time is:

$$p_{2\oplus 3} = \frac{n_2 - 1}{n_2 n_3 - 1} p_2 + \frac{n_2 n_3 - n_2}{n_2 n_3 - 1} p_3 \tag{4.7}$$

A remote acquisition, a few local passes, and a remote release sequence repeats in each domain. The C-MCS$_{\text{in}}$ lock acquires c_{in} number of locks in time $p_{2\oplus 3} + (c_{in} - 1)p_1$. Hence the throughput of a C-MCS$_{\text{in}}$ lock, $\mathcal{T}_{in}$, in a 3-level system is given by:

$$\mathcal{T}_{in}(3) = \frac{c_{in}}{p_{2\oplus 3} + (c_{in} - 1)p_1} \tag{4.8}$$

As $c_{in} \to \infty$, $\mathcal{T}_{in}(3) \to 1/p_1$. Maximum throughput of the C-MCS$_{\text{in}}$ lock, $\mathcal{T}_{in}^{max}$, in a 3-level system is given by:

$$\mathcal{T}_{in}^{max}(3) = \frac{1}{p_1} \tag{4.9}$$

This bound holds for any N-level system.

4.6.1.2 Throughput of the C-MCS$_{\text{out}}$ lock

In this lock, cohorts are formed among threads belonging to level-1 and level-2 NUMA domains aggregated, and no distinction is made between these two levels of hierarchy. Let c_{out} be the passing threshold. Let the cost of acquisition and release, from an outside domain (at level-3), be Acq_3 and Rel_3 respectively. We represent $Acq_3 + Rel_3$ as p_3 — the lock passing time at level 3. The expected cohort passing time within level-1 and level-2 NUMA domains is:

$$p_{1\oplus 2} = Pr[\textit{passing within level 1}]\ p_1 + (1 - Pr[\textit{passing within level 1}])\ p_2$$

There are $(n_1 n_2 - 1)$ other threads within level-2 of the NUMA hierarchy, of which $(n_1 - 1)$ are level-1 NUMA peers. Hence, the probability of cohort passing between level-1 NUMA peers is $(n_1 - 1)/(n_1 n_2 - 1)$.

$$\implies p_{1\oplus 2} = \frac{n_1 - 1}{n_1 n_2 - 1} p_1 + \frac{n_1 n_2 - n_1}{n_1 n_2 - 1} p_2 \tag{4.10}$$

The C-MCS$_{\text{out}}$ lock passes the lock $(c_{out} - 1)$ times within its cohorts adding a total of $(c_{out} - 1)p_{1\oplus 2}$ length to the critical path. c_{out} lock acquisitions happen in time $p_3 + (c_{out} - 1)p_{1\oplus 2}$. The sequence—*a remote acquisition at level-3, a few local passing within*

either level-1 or level-2, and a remote release to level-3—repeats in each domain. Hence, throughput of the C-MCS$_{\text{out}}$ lock, $\mathcal{T}_{out}$, in a 3-level system is given by:

$$\mathcal{T}_{out}(3) = \frac{c_{out}}{p_3 + (c_{out} - 1)p_{1\oplus 2}} \tag{4.11}$$

As $c_{out} \to \infty$, $\mathcal{T}_{out} \to 1/p_{1\oplus 2}$. Maximum throughput of a C-MCS$_{\text{out}}$ lock, $\mathcal{T}_{out}^{max}$, in a 3-level system is given by:

$$\mathcal{T}_{out}^{max}(3) = \frac{1}{p_{1\oplus 2}} \tag{4.12}$$

This bound holds for any N-level system.

In general, if the innermost cohort formation is at level k, the peak throughput is bounded by $1/p_{1\oplus 2\oplus \ldots \oplus k}$.

Since p_2 (the lock passing time at level 2) is higher than p_1 (the lock passing time at level 1), $p_{1\oplus 2}$ (the expected lock passing time between level-1 or level-2) is higher than p_1. From Eqn 4.9 and 4.12, it follows that $\mathcal{T}_{out}^{max}(3) < \mathcal{T}_{in}^{max}(3)$. Hence, *C-MCS$_{in}$ achieves higher peak throughput than C-MCS$_{out}$*. Experiments on an IBM Power 755, a 3-level system, corroborate this finding (Sec. 4.8).

4.6.1.3 Throughput of HMCS locks

The HMCS⟨3⟩ lock forms cohorts at both level 1 and level 2 of the NUMA hierarchy in a 3-level system (Figure 4.7). Let h_1 and h_2 respectively be the passing thresholds at level 1 and level 2. In a 3-level lock, a full cycle begins by obtaining the lock at level-3 (Acq_3). Subsequently, the lock is passed $(h_1 - 1)$ times within level-1, releasing the lock at level-2 (Rel_2), followed by an acquisition at level-2 (Acq_2). The cycle of passing within level-2 happens for $(h_2 - 1)$ times. On reaching the passing threshold at level 2, the HMCS⟨3⟩ lock relinquishes the global lock to a level-3 peer adding Rel_3 to the critical path. There will be a total of $(h_1 - 1)h_2$ lock passes within level 1, each one adding p_1 length to the critical path. There will be $(h_2 - 1)$ lock passes at level 2, each one adding $\hat{p}_2$ length to the critical path. There will be one lock passing at level 3 adding $\hat{p}_3$ length to the critical path. In the entire cycle, $h_1 h_2$ locks will be acquired. Hence, throughput of the HMCS⟨3⟩ lock, $\mathcal{T}_{hmcs\langle 3\rangle}$, in a 3-level system is given by:

$$\mathcal{T}_{hmcs\langle 3\rangle}(3) = \frac{h_1 h_2}{\hat{p}_3 + (h_2 - 1)\hat{p}_2 + h_2(h_1 - 1)p_1} \tag{4.13}$$

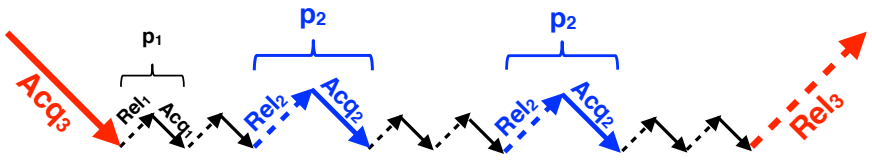


Figure 4.7 : Lock passing in the HMCS⟨3⟩ lock.

$\hat{p}_3$ and $\hat{p}_2$ are the lock passing times respectively at level-2 and level-3 in the HMCS⟨3⟩ lock. The level-2 lock passing in the HMCS⟨3⟩ lock has an additional critical path component that loads and compares the status flag (lines 26-28 in Listing 4.1) with the *threshold*, which is not needed by optimal implementations of C-MCS locks. If this additional cost is ϵ_0, then $\hat{p}_2 = p_2 + \epsilon_0$. On reaching the passing threshold at level 2, the HMCS⟨3⟩ lock needs to traverse to the parent lock before releasing the global lock to a level-3 peer (line 30 in Listing 4.1)—a cost not incurred in C-MCS locks. Let ϵ_1 be the cost of traversing from level-2 lock to the level-3 parent lock (all necessary datum will be in the nearest level cache). For that reason, $\hat{p}_3 = p_3 + \epsilon_1$. Hence, the throughput of the HMCS⟨3⟩ lock is:

$$\mathcal{T}_{hmcs\langle 3\rangle}(3) = \frac{h_1 h_2}{p_3 + (h_2 - 1)p_2 + h_2(h_1 - 1)p_1 + \epsilon} \tag{4.14}$$

where $\epsilon = \epsilon_1 + (h_2 - 1)\epsilon_0$.

As $h_1 \to \infty$, $\mathcal{T}_{hmcs\langle 3\rangle} \to 1/p_1$. Maximum throughput of the HMCS⟨3⟩ lock, $\mathcal{T}^{max}_{hmcs\langle 3\rangle}$, in a 3-level system is given by:

$$\mathcal{T}^{max}_{hmcs\langle 3\rangle}(3) = \frac{1}{p_1} \tag{4.15}$$

It is straightforward to infer from Eqn 4.9, 4.12 and 4.15 that:

$$\mathcal{T}^{max}_{in}(3) = \mathcal{T}^{max}_{hmcs\langle 3\rangle}(3) > \mathcal{T}^{max}_{out}(3) \tag{4.16}$$

Hence, *the C-MCS$_{in}$ lock—a C-MCS variant with inner cohorting and the HMCS⟨3⟩ lock have the same peak throughput and both of them can deliver higher throughput than Dice et al.'s C-MCS lock with outer cohorting.*

From Eqn 4.13, it is straightforward to generalize the throughput of an N-level HMCS

lock, $\mathcal{T}_{hmcs\langle N\rangle}$, for an N-level NUMA system as:

$$\mathcal{T}_{hmcs\langle N\rangle}(N) = \frac{\prod_{i=1}^{N-1} h_i}{p_N + \sum_{i=1}^{N-1} \left(p_i(h_i - 1) \prod_{j=i+1}^{N-1} h_j \right)} \tag{4.17}$$

Discussion: While our performance models assumed empty critical sections for assessing lock overhead, we can model non-empty critical sections by including an additional *cs* term to account for the cost of the critical section in each lock passing term (p_i). While the C-MCS$_{\text{in}}$ and HMCS locks can theoretically deliver top throughput, in practice, threads may not be able to achieve the maximum possible throughput when the delay outside the critical path is large. The average number of threads enqueuing for the same lock acquisition at a level per unit time is defined as the *arrival rate* at that level. For a lock representing a NUMA domain in the HMCS tree to be useful, the arrival rate for that domain must exceed the rate at which the domain is selected for service. Furthermore, the reduction in access latency realized by passing a lock and the data it protects within a domain should outweigh the cost of adding an addition lock level.

4.6.2 (Un)Fairness

If a lock serves threads in the first-in first-out (FIFO) order, then the lock is considered fully fair. The MCS lock ensures FIFO ordering once a requesting thread has swapped the lock's tail pointer. HMCS and C-MCS locks do not ensure FIFO ordering since they allow multiple turns for threads within the same domain even when threads are already enqueued elsewhere in the lock hierarchy.

For quantitative comparison of HMCS and C-MCS locks, we devise a metric of *unfairness*. In this scheme, *after a process has made a request to enter its critical section and before that request is granted, each subdomain can acquire the lock as many times as the total number of threads in its subdomain; every additional acquisition counts towards a unit of unfairness.* Under high contention, a thread that just released the lock must wait until all other threads already waiting at that level are served, which is ensured by the FIFO property of MCS lock at each level. Hence, from the viewpoint of a waiting thread, this scheme penalizes a thread

for multiple acquisitions, but it does not penalize for the non-FIFO order of servicing the requests.

Definition 4.3 (Unfairness:) Unfairness *is the maximum possible total number of additional rounds of lock acquisitions over the designated quota of one, by other threads in the system when a thread is still waiting to acquire the lock. Formally, if T is the set of all threads in the system, the unfairness, $\mathcal{U}$, is given by:*
$\mathcal{U} = Max\Big(\forall j \in T, \sum_{i \in T-j} \big(\ acquisitions\ by\ i \mid j\ is\ waiting - 1\big)\Big)$.

The worst-case wait for a thread when using an MCS lock is $(\prod_{i=1}^{N} n_i - 1)$ lock acquisitions and its unfairness is 0. In fully contended cohort locks, if the threshold of each level h_i equals to the number of contenders n_i at level i, then the longest waiting thread waits for a maximum of $(\prod_{i=1}^{N} n_i - 1)$ lock acquisition, which is same as the longest wait in the classical MCS lock. However, if any $h_i > n_i$, threads will be served more than once causing unfairness for some waiting threads in the system.

4.6.2.1 Unfairness of the C-MCS$_{\text{in}}$ lock

Consider the case when $c_{in} \geq n_1$. The maximum unfairness will be incurred by the last thread— $t_{4.2}$ shown in blue color in Figure 4.8, enqueued in the last domain enqueued (domain 4) in the outer queue of the C-MCS$_{\text{in}}$ lock. Each of the first $(n_2 n_3 - 1)$ NUMA domains acquires c_{in} locks, of which only n_1 are fair and the remaining $(c_{in} - n_1)$ are unfair. The acquisitions in the last enqueued domain do not contribute to the unfairness since by the time repetitions happen, the longest waiting thread would have already been served. Hence the unfairness of the C-MCS$_{\text{in}}$ lock in a 3-level system is:

$$\mathcal{U}_{in}(3) = (c_{in} - n_1)(n_2 n_3 - 1) \tag{4.18}$$

In Figure 4.8, $n_1 = 2, n_2 = 2, n_3 = 2$, and $c_{in} = 4$. Hence, the total unfairness is $(4 - 2)(2 \times 2 - 1) = 6$.

When $c_{in} < n_1$, each domain goes through multiple rounds of service before the longest waiting thread in the last domain can be served. We demonstrate this with the example in Figure 4.9, where $n_1 = 4, n_2 = 2, n_3 = 2$, and $c_{in} = 3$. Only 3 out of 4 threads will be

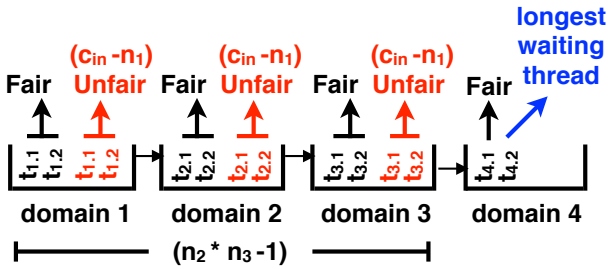


Figure 4.8 : Unfairness in the C-MCS$_{\text{in}}$ lock, when $c_{in} \geq n_1$.

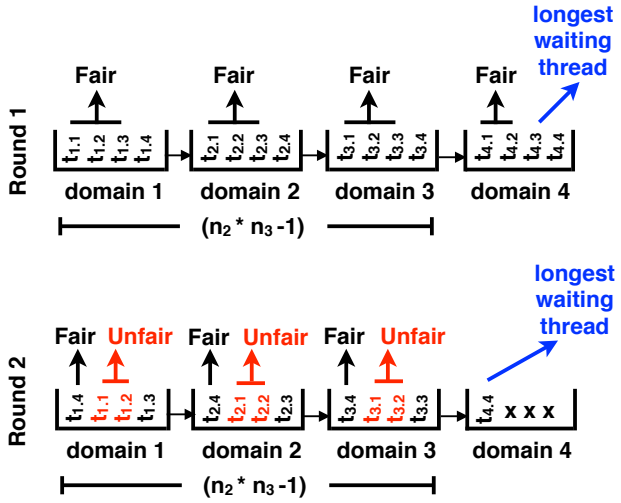


Figure 4.9 : Unfairness in the C-MCS$_{\text{in}}$ lock, when $c_{in} < n_1$.

served in each domain in the first round through all domains. For example, in domain 1, $t_{1.1}$, $t_{1.2}$, and $t_{1.3}$ will be served and $t_{1.4}$ will not be served. In the worst case, it takes a total of two rounds through all domains before the longest waiting thread can be served. In the second round through all the domains, the first thread in domains 1-3 ($t_{1.4}$, $t_{2.4}$, and $t_{3.4}$, respectively) will be served for the first time. In this round, $t_{1.1}$ and $t_{1.2}$, will be taking their additional rounds, adding two units of unfairness in domain *1*. Similarly, domains *2* and *3* also contribute two units of unfairness each. As before, no repetitions happen in the last domain. Each domain gets lock acquisitions in c_{in} quantum; from the point of view of longest waiting thread, each of the other $(n_2 n_3 - 1)$ domains acquire $\lceil \frac{n_1}{c_{in}} \rceil c_{in}$ locks, of which n_1 are fair. Hence, the total unfairness is:

$$\mathcal{U}_{in}(3) = \left(\left\lceil \frac{n_1}{c_{in}} \right\rceil c_{in} - n_1 \right) (n_2 n_3 - 1) \tag{4.19}$$

When $c_{in} \geq n_1$, Eqn 4.19 simply reduces into Eqn 4.18. In Figure 4.9, the unfairness is

$(\lceil 4/3 \rceil 3 - 4)(2 \times 2 - 1) = 6.$

4.6.2.2 Unfairness of the C-MCS$_{\text{out}}$ lock

The C-MCS$_{\text{out}}$ lock collapses level-1 and level-2 of the NUMA hierarchy into a single domain of $n_1 n_2$ threads and assumes that there are n_3 such domains. Following the same argument as before, it is straightforward to show that the unfairness of the C-MCS$_{\text{out}}$ lock in a 3-level system is:

$$\mathcal{U}_{out}(3) = \left(\left\lceil \frac{n_1 n_2}{c_{out}} \right\rceil c_{out} - n_1 n_2 \right)(n_3 - 1) \tag{4.20}$$

4.6.2.3 Unfairness of HMCS locks

First we compute the unfairness when, $h_1 \geq n_1$ and $h_2 \geq n_2$.

Consider a system where $n_1 = 2, n_2 = 2$ and let $h_1 = 4, h_2 = 4$ as shown in Figure 4.10. The maximum unfairness will be observed by thread $t_{M.2.2}$. There will be 4×4 lock acquisitions inside each of the first $(n_3 - 1)$ level-2 domains, of which $(4 \times 4 - 2 \times 2)$ are unfair. Inside the last domain (M), there will be $(4 - 2)$ unfair acquisitions in the subdomain $M.1$. The subdomain $M.2$ adds no unfairness.

In general, maximum unfairness will be observed by the last enqueued thread in the last level-1 domain ($D_{last_{L1}}$) belonging to the last enqueued level-2 domain ($D_{last_{L2}}$). There will be $h_1 h_2$ lock acquisitions inside each of the first $(n_3 - 1)$ level-2 domains of which $(h_1 h_2 - n_1 n_2)$ are unfair. Inside the $D_{last_{L2}}$ domain, there will be $(h_1 - n_1)$ unfair acquisitions in each of the first $(n_2 - 1)$ level-1 domains. The $D_{last_{L1}}$ domain adds no unfairness since any repetitions will happen only after the longest-waiting thread is already served.

Thus the total unfairness of the HMCS$\langle 3 \rangle$ lock in a 3-level system when $h_1 \geq n_1$ and $h_2 \geq n_2$ is:

$$\mathcal{U}_{hmcs\langle 3 \rangle}(3) = (h_1 h_2 - n_1 n_2)(n_3 - 1) + (h_1 - n_1)(n_2 - 1) \tag{4.21}$$

We use Figure 4.11 to provide intuition for deriving unfairness when $h_1 < n_1$ and $h_2 < n_2$. In Figure 4.11, $n_1 = 3$, $n_2 = 4$, $n_3 = 2$, $h_1 = 2$, and $h_2 = 3$. In the first round, three (domains 1.1, 1.2, and 1.3) out of four level-1 domains will be served inside domains 1 and 2. Within each level-1 domain that is served, two out of three threads will be served (e.g., $t_{1.1.1}$ and $t_{1.1.2}$ will be served in domain 1.1 but not $t_{1.1.3}$). Round 1 accrues no unfairness.

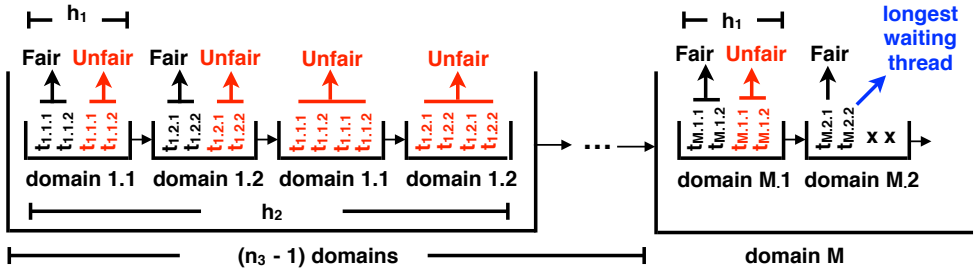


Figure 4.10 : Unfairness in the HMCS⟨3⟩ lock, when $h_1 \geq n_1$ and $h_2 \geq n_2$.

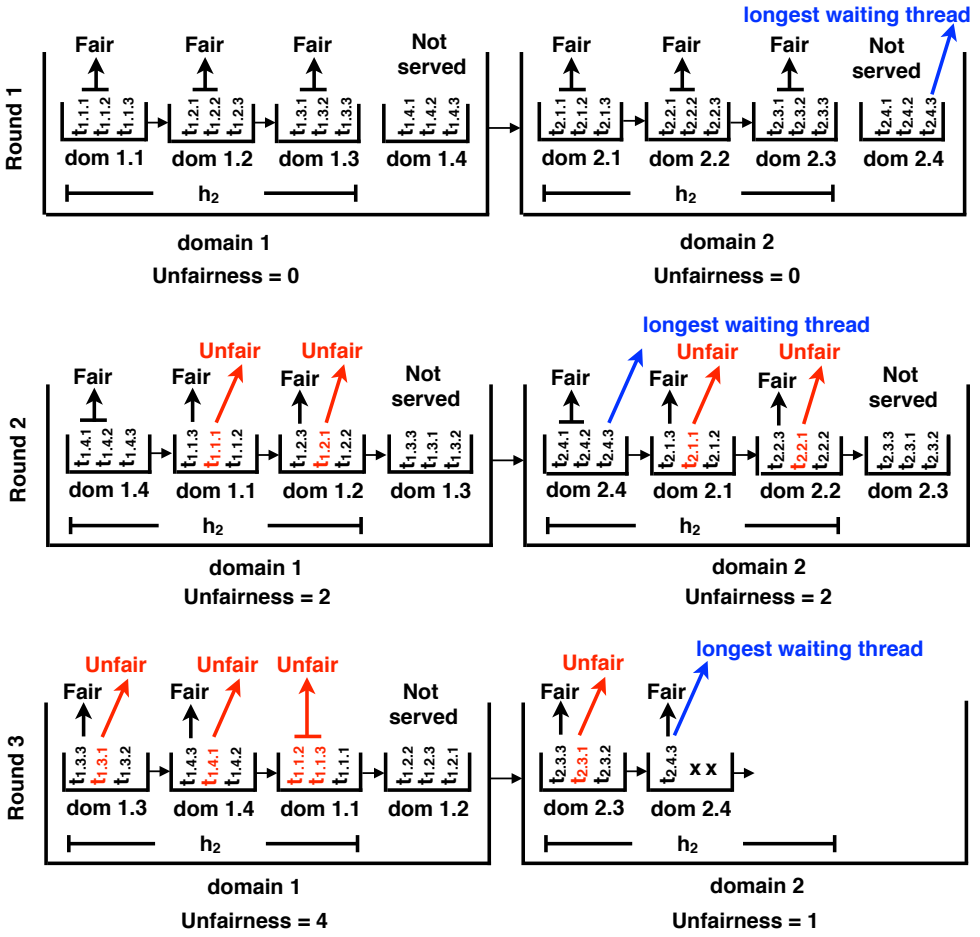


Figure 4.11 : Unfairness in the HMCS⟨3⟩ lock, when $h_1 < n_1$ and $h_2 < n_2$.

In round 2, two threads ($t_{1.4.1}$ and $t_{1.4.2}$) inside the domain *1.4* will be served for the first time. In domains *1.1* and *1.2* one thread ($t_{1.1.3}$ and $t_{1.2.3}$, respectively) will be served for the first time, whereas one thread ($t_{1.1.1}$ and $t_{1.2.1}$, respectively) will be enjoying its second round of service, while the longest waiting thread is still waiting. This accumulates an unfairness of 2 units in domains 1. The same repeats in domain-2 also. In the third

round, one thread ($t_{1.3.3}$ and $t_{1.4.3}$, respectively) in the domains *1.3* and *1.4* will be served fairly, whereas one thread ($t_{1.3.1}$ and $t_{1.4.1}$, respectively) will be served for the second time, hence unfair. Furthermore, because of a remaining one round in the h_2 quantum, we serve two threads ($t_{1.1.2}$ and $t_{1.1.3}$) in the domain *1.1* for the 3^{rd} time accruing a total of 4 units of unfairness in domain *1*. In domain *2.3*, one thread ($t_{2.3.3}$) is served fairly, whereas one thread ($t_{2.3.1}$) will be served for the second time. Thus, total unfairness in round 3 is 5. The total unfairness in all rounds taken together is 9.

Since each level-2 lock acquisition provides a chunk of h_1 locks at a time, each level-1 domains has to experience $\lceil \frac{n_1}{h_1} \rceil$ level-2 lock acquisitions before the longest-waiting thread is served. Let us first focus on a level-2 domain to which the longest waiting thread does not belong to arrive at the number of level-3 acquisitions that need to happen. Since there are n_2 peers at level-2, a total of $\lceil \frac{n_1}{h_1} \rceil n_2$ level-2 lock acquisitions need to happen. Level-2 locks are given in a chunk of h_2, which means each level-2 domain needs to experience $\lceil \lceil \frac{n_1}{h_1} \rceil \frac{n_2}{h_2} \rceil$ level-3 lock acquisitions. Each round of level-3 lock acquisition serves a total of $h_1 h_2$ locks inside its subdomain. Hence, the total lock acquisitions will be $\lceil \lceil \lceil \frac{n_1}{h_1} \rceil \frac{n_2}{h_2} \rceil h_1 h_2$, of which only $n_1 n_2$ are fair. Hence, the unfairness in each domain to which the longest waiting thread does not belong is $\lceil \lceil \lceil \frac{n_1}{h_1} \rceil \frac{n_2}{h_2} \rceil h_1 h_2 - n_1 n_2$. There is a maximum of $(n_3 - 1)$ such domains.

In the level-2 domain to which the longest-waiting thread belongs, each of the $(n_2 - 1)$ domains need to experience $\lceil \frac{n_1}{h_1} \rceil h_1$ lock acquisitions, of which only n_1 will be fair. Thus, the total unfairness in the last level-2 domain will be $(\lceil \frac{n_1}{h_1} \rceil h_1 - n_1)(n_2 - 1)$. Combining the unfairness from both cases, we arrive at the total unfairness of the HMCS$\langle 3 \rangle$ lock in a 3-level system as:

$$\mathcal{U}_{hmcs\langle 3 \rangle}(3) = \left(\left\lceil \left\lceil \frac{n_1}{h_1} \right\rceil \frac{n_2}{h_2} \right\rceil h_1 h_2 - n_1 n_2 \right)(n_3 - 1) + \left(\left\lceil \frac{n_1}{h_1} \right\rceil h_1 - n_1 \right)(n_2 - 1) \tag{4.22}$$

For the example in Figure 4.11, the unfairness is:

$$(\lceil \lceil 3/2 \rceil 4/3 \rceil 2 \times 3 - 3 \times 4)(2 - 1) + (\lceil 3/2 \rceil 2 - 3)(4 - 1) = 9.$$

When $h_1 \geq n_1$ and $h_2 \geq n_2$, Eqn 4.22 degenerates into Eqn 4.21. Eqn 4.21 also covers the cases of $h_1 \geq n_1$ and $h_2 < n_2$, as well as $h_1 < n_1$ and $h_2 \geq n_2$. We omit the details for brevity. We note that when $h_1 < n_1$ unfairness is *0*, if either h_1 divides n_1 and h_2 divides n_2 or h_2 divides $\lceil \frac{n_1}{h_1} \rceil n_2$.

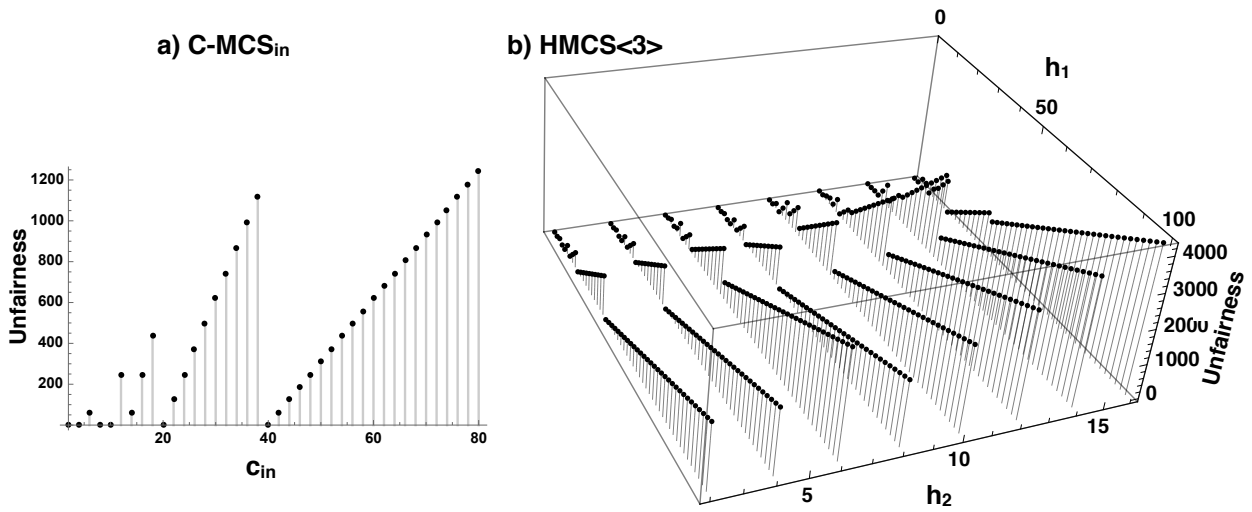


Figure 4.12 : Impact of threshold on unfairness.

Figure 4.12 provides visualization of unfairness in the C-MCS$_{\text{in}}$ and HMCS$\langle 3 \rangle$ locks with varying values of thresholds. Both graphs assume a hypothetical machine where $n_1 = 40, n_2 = 8$, and $n_3 = 4$. For the C-MCS$_{\text{in}}$ lock, the unfairness is 0, when the threshold c_{in} divides n_1. For the HMCS$\langle 3 \rangle$ lock, the unfairness is 0, when the threshold h_1 divides n_1 and h_2 divides n_2; otherwise, the unfairness grows linearly with both increase in h_1 and h_2. Finally, we extend the Eqn 4.22 to generalize the unfairness of an N-level HMCS lock, $\mathcal{U}_{hmcs\langle N \rangle}$, for any N-level NUMA system as:

$$\mathcal{U}_{hmcs\langle N \rangle}(N) = \sum_{i=1}^{N-1} \Big(\psi_i \prod_{j=1}^{i} h_i - \prod_{j=1}^{i} n_i \Big)(n_{i+1} - 1) \tag{4.23}$$

$$\text{where } \psi_i = \left\lceil \left\lceil \left\lceil \frac{n_1}{h_1} \right\rceil \frac{n_2}{h_2} \right\rceil \ldots \frac{n_i}{h_i} \right\rceil$$

When $h_i \geq n_i, \forall i$, the ψ_i term in Eqn 4.23 vanishes.

4.7 HMCS properties

In this section, we prove some of the important attributes of HMCS locks in comparison with the C-MCS locks. From Eqn 4.16 the peak throughput of the HMCS lock is same as the peak throughput of the C-MCS$_{\text{in}}$ lock. Hence, the competition between them is for fairness when delivering the same throughput. We prove, in Section 4.7.1, that on a 3-level NUMA system, when the difference in latencies between two consecutive levels of the NUMA hierarchy is sufficiently large, a 3-level HMCS lock delivers higher fairness than a C-MCS$_{\text{in}}$ lock for any user-chosen throughput.

The competition between the C-MCS$_{\text{out}}$ and HMCS$\langle 3 \rangle$ locks is for the throughput at the same level of fairness. In this regard, in Section 4.7.2, we prove that on a 3-level system, when the difference in latencies between two consecutive levels of NUMA hierarchy is sufficiently large, a 3-level HMCS lock delivers higher throughput than the C-MCS$_{\text{out}}$ lock for any user-chosen fairness level.

4.7.1 Fairness assurance of HMCS over C-MCS$_{\text{in}}$

Let $\alpha \in [0 - 1]$ be the user chosen level of throughput represented as a fraction of peak throughput. If the value of c_{in} in the C-MCS$_{\text{in}}$ lock needed to achieve this throughput is less than n_1, then, the HMCS$\langle 3 \rangle$ lock can set $h_1 = n_1$ and $h_2 = n_2$, which not only achieves more throughput but also delivers 0 unfairness, thus proving its superiority. Hence, the comparison is needed only when $c_{in} \geq n_1$. The value of c_{in} to achieve the expected throughput is derived by solving Eqn 4.8:

$$\begin{aligned} \frac{c_{in}}{p_{2 \oplus 3} + (c_{in} - 1)p_1} &= \alpha \, \mathcal{T}_{in}^{max}(3) \\ \implies c_{in} &= \frac{\alpha(p_{2 \oplus 3} - p_1)}{p_1(1 - \alpha)} \end{aligned} \tag{4.24}$$

In the HMCS$\langle 3 \rangle$ lock we set $h_2 = n_2$. The value of h_1 to achieve the expected throughput α

is derived by solving Eqn 4.14.

$$\frac{h_1 n_2}{p_3 + (n_2 - 1)p_2 + n_2(h_1 - 1)p_1 + \epsilon} = \alpha\ \mathcal{T}^{max}_{hmcs\langle 3\rangle}(3)$$
$$\implies h_1 = \frac{\alpha(p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon)}{n_2 p_1(1 - \alpha)} \tag{4.25}$$

Again, only values of $h_1 \geq n_1$ are of interests since smaller values can be replaced with $h_1 = n_1$ to obtain 0 unfairness but superior throughput. We note in passing that practical comparisons can only be made when the thresholds in both Eqn 4.24 and 4.25 take integer values. Substituting c_{in} from Eqn 4.24 in Eqn 4.18, we get:

$$\mathcal{U}_{in}(3) = \Big(\frac{\alpha(p_{2\oplus 3} - p_1)}{p_1(1 - \alpha)} - n_1\Big)(n_2 n_3 - 1) \tag{4.26}$$

Substituting $h_2 = n_2$ and h_1 from Eqn 4.25 in Eqn 4.21, we get:

$$\mathcal{U}_{hmcs\langle 3\rangle}(3) = \Big(\frac{\alpha(p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon)}{n_2 p_1(1 - \alpha)} - n_1\Big)(n_2 n_3 - 1) \tag{4.27}$$

From Eqn 4.26 and 4.27, the necessary condition for the HMCS$\langle 3\rangle$ lock to deliver higher fairness than the C-MCS$_{\text{in}}$ lock is:

$$\begin{aligned}
& \textit{Eqn 4.26} && \geq \textit{Eqn 4.27} \\
\implies & \Big(\frac{\alpha(p_{2\oplus 3} - p_1)}{p_1(1 - \alpha)} - n_1\Big)(n_2 n_3 - 1) && \geq \Big(\frac{\alpha(p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon)}{n_2 p_1(1 - \alpha)} - n_1\Big)(n_2 n_3 - 1) \\
\implies & p_{2\oplus 3} - p_1 && \geq \frac{p_3 + p_2(n_2 - 1) - p_1 n_2 + \epsilon}{n_2} \\
\implies & n_2 p_{2\oplus 3} && \geq p_3 + p_2(n_2 - 1) + \epsilon
\end{aligned} \tag{4.28}$$

Substituting the value of $p_{2\oplus 3}$ from Eqn 4.7 in Eqn 4.28, we get:

$$\begin{aligned} & n_2\Big(\frac{n_2-1}{n_2n_3-1}p_2+\frac{n_2n_3-n_2}{n_2n_3-1}p_3\Big) && \geq p_3+p_2(n_2-1)+\epsilon \\ \implies & p_3\Big(\frac{n_2^2n_3-n_2^2-n_2n_3+1}{n_2n_3-1}\Big)+p_2(n_2-1)\Big(\frac{n_2-n_2n_3+1}{n_2n_3-1}\Big) && \geq \epsilon \\ \implies & p_3\Big(\frac{n_2n_3(n_2-1)-(n_2^2-1)}{n_2n_3-1}\Big)+p_2(n_2-1)\Big(\frac{n_2-n_2n_3+1}{n_2n_3-1}\Big) && \geq \epsilon \end{aligned} \tag{4.29}$$

Substituting $n_2^2-1=(n_2-1)(n_2+1)$ in the LHS of Eqn 4.29, we get:

$$\begin{aligned} & p_3(n_2-1)\Big(\frac{n_2n_3-n_2-1)}{n_2n_3-1}\Big)-p_2(n_2-1)\Big(\frac{n_2n_3-n_2-1}{n_2n_3-1}\Big) && \geq \epsilon \\ \implies & \frac{(p_3-p_2)(n_2-1)(n_2n_3-n_2-1)}{n_2n_3-1} && \geq \epsilon \end{aligned} \tag{4.30}$$

Since $h_2=n_2$, we can substitute $\epsilon=(n_2-1)\epsilon_0+\epsilon_1$ in the RHS of Eqn 4.30, and we get:

$$\begin{aligned} & \frac{(p_3-p_2)(n_2-1)(n_2n_3-n_2-1)}{n_2n_3-1} && \geq (n_2-1)\epsilon_0+\epsilon_1 \\ \implies & (p_3-p_2)\Big(1-\frac{n_2}{n_2n_3-1}\Big) && \geq \epsilon_0+\frac{\epsilon_1}{n_2-1} \end{aligned} \tag{4.31}$$

All terms on the LHS and RHS are positive in Eqn 4.31. On any NUMA system, $n_1, n_2, n_3 \geq 2$; otherwise it does not form a new NUMA domain at a level. It can be shown that the minimum value of the second term on LHS is 1/3. Hence, the sufficiency condition for the HMCS$\langle 3\rangle$ lock to deliver higher fairness than the C-MCS$_{\text{in}}$ lock at the same throughput is:

$$\frac{(p_3-p_2)}{3} \geq \epsilon_0+\frac{\epsilon_1}{n_2-1} \tag{4.32}$$

From Eqn 4.31 and Eqn 4.32 we make the following observations:

1. As long as the cost of additional check of status flag and the amortized cost of traversing to parent level lock once in $h_2=n_2$ quantum is less than one third the difference in passing time between level-3 and level-2, it calls for having an additional level in the

HMCS lock hierarchy.

2. Larger difference in passing time (also related to the access latency) between two consecutive levels in the NUMA hierarchy (first term in LHS of Eqn 4.31) governs the necessity for additional level in the HMCS lock hierarchy.
3. From the second term in LHS of Eqn 4.31, it follows that if a NUMA domain d has many subdomains, then one can benefit from having an HMCS lock for each NUMA subdomain of d.

4.7.2 Throughput assurance of HMCS over C-MCS$_{\text{out}}$

Let k be the threshold value of the C-MCS$_{\text{out}}$ lock that achieves the user chosen unfairness value of β. Then from Eqn 4.20, we know that

$$\mathcal{U}_{out}(3) = \left(\left\lceil \frac{n_1 n_2}{k} \right\rceil k - n_1 n_2\right)(n_3 - 1) \tag{4.33}$$

We set $h_1 = n_1$ and $h_2 = k/n_1$ in the HMCS$\langle 3 \rangle$ lock. The concern here is whether a fractional h_2 value is possible. The answer is yes. For example, assume $n_1 = 4$, $n_2 = 3$, and $k = 13$. We want to set the following configuration: $h_1 = n_1 = 4$, $h_2 = 13/4 = 3.25$. The HMCS$\langle 3 \rangle$ lock can hand out 3 rounds of level-2 quanta each containing $h_1 (= 4)$ lock acquisitions, totaling $3 \times 4 = 12$ acquisitions. But in the 4^{th} round, the level-2 lock should curtail level-1's threshold from $n_1 = 4$ to $0.25 \times n_1 = 0.25 \times 4 = 1$. Notice that $0.25 \times 4 = 1 = 13 \ mod \ 4 = k \ mod \ n_1$. With this, we would have given out $12 + 1 = 13 = k$ acquisitions with exactly the same amount of unfairness as the C-MCS$_{\text{out}}$ lock. It is no accident that for the last round we chose $k \ mod \ n_1$ threshold; it simply follows from algebra that $\lfloor \frac{k}{n_1} \rfloor n_1 + k \ mod \ n_1 = k$.

Intuitively, since level-1 locks go in a quantum of size n_1, we want to curtail the last quantum to a smaller value, i.e., $k \ mod \ n_1$. With this knowledge, we can slightly alter the acquire protocol. Instead of starting the count from 1 till n_1 to reach the threshold, we start the counter (the `status` field of the `QNode`) from $(k - k \ mod \ n_1)$ for the last round of level-1 acquisitions when there is need to achieve the fractional h_2 value. This modification adds one extra compare on the critical path, but we note that this modification is only of theoretical interest to demonstrate that the fractional h_2 value is achievable.

Substituting $h_1 = n_1$ and $h_2 = k/n_1$ in Eqn 4.22 yields:

$$\begin{aligned}\mathcal{U}_{hmcs}(3) &= \left(\left\lceil \left\lceil \frac{n_1}{n_1} \right\rceil \frac{n_1 n_2}{k} \right\rceil n_1 \frac{k}{n_1} - n_1 n_2\right)(n_3 - 1) + \left(\left\lceil \frac{n_1}{n_1} \right\rceil n_1 - n_1\right)(n_2 - 1) \\ &= \left(\left\lceil \frac{n_1 n_2}{k} \right\rceil k - n_1 n_2\right)(n_3 - 1) \\ &= Eqn\ 4.33\end{aligned}$$

Thus, when $h_1 = n_1$ and $h_2 = k/n_1$, C-MCS$_{\text{out}}$ and HMCS$\langle 3 \rangle$ locks deliver the same fairness.

Now, we derive the conditions for $\mathcal{T}_{hmcs\langle 3\rangle}(3)$ to be greater than $\mathcal{T}_{out}(3)$ in this configuration. Substituting $c_{out} = k$ in the throughput equation for the C-MCS$_{\text{out}}$ lock (Eqn 4.11), we get:

$$\mathcal{T}_{out}(3) = \frac{k}{p_3 + (k-1)p_{1\oplus 2}} \tag{4.34}$$

Substituting $h_1 = n_1$ and $h_2 = k/n_1$, in the throughput equation for the HMCS$\langle 3 \rangle$ lock (Eqn 4.14), we get:

$$\mathcal{T}_{hmcs\langle 3\rangle}(3) = \frac{n_1 \frac{k}{n_1}}{p_3 + (\frac{k}{n_1} - 1)p_2 + \frac{k}{n_1}(n_1 - 1)p_1 + \epsilon} \tag{4.35}$$

From Eqn 4.34 and 4.35, to show that $\mathcal{T}_{hmcs}(3) \geq \mathcal{T}_{out}(3)$, we simply need to show that:

$$(k-1)p_{1\oplus 2} \geq \ (\frac{k}{n_1} - 1)p_2 + \frac{k}{n_1}(n_1 - 1)p_1 + \epsilon \tag{4.36}$$

Substituting for $p_{1\oplus 2}$ from Eqn 4.10 in Eqn 4.36, we obtain:

$$(p_2 - p_1)\left(\frac{k(n_1 n_2 - n_1 - n_2) + (n_1 - 1)}{(n_1 n_2 - 1)(\frac{k}{n_1} - 1)} + \frac{k}{n_1(n_1 n_2 - 1)(\frac{k}{n_1} - 1)}\right) \ \geq \ \epsilon_0 + \frac{\epsilon_1}{\frac{k}{n_1} - 1} \tag{4.37}$$

Eqn 4.37 forms the basic constraint to ensure that the HMCS$\langle 3 \rangle$ lock has higher throughput compared to the C-MCS$_{\text{out}}$ lock. Each term on the LHS and RHS of Eqn 4.37 is positive. When k increases, the LHS increases and RHS decreases, ensuring that beyond a certain value of k, the inequality is always true. Hence, we need to find the sufficiency condition at the smallest value that k can assume. Any value of $k < n_1 n_2$ increases unfairness in both locks and decreases the throughput. Hence, if the C-MCS$_{\text{out}}$ chooses $k < n_1 n_2$, for the HMCS$\langle 3 \rangle$ lock we simply choose $h_1 = n_1, h_2 = n_2$, which guarantees no unfairness and

yet delivers higher throughput than the C-MCS$_{\text{out}}$ lock. Hence, the smallest value that a C-MCS$_{\text{out}}$ lock can choose for k is $n_1 n_2$. Substituting $k = n_1 n_2$ in Eqn 4.37, we arrive at

$$(p_2 - p_1)\left(\frac{n_1 n_2(n_1 n_2 - n_1 - n_2) + (n_1 - 1) + n_2}{(n_1 n_2 - 1)(n_2 - 1)}\right) \geq \epsilon_0 + \frac{\epsilon_1}{n_2 - 1} \tag{4.38}$$

On any system, $n_1, n_2, n_3 \geq 2$, otherwise it does not form a new NUMA domain at that level. It can be shown that the minimum value of the second term on LHS in Eqn 4.38 is 3. Hence the sufficiency condition for the HMCS$\langle 3 \rangle$ lock to deliver higher throughput than the C-MCS$_{\text{out}}$ lock at the same fairness level is:

$$(p_2 - p_1)(3) \geq \epsilon_0 + \frac{\epsilon_1}{n_2 - 1} \tag{4.39}$$

From Eqn 4.37, 4.38, and 4.39 we make the following observations:

1. There is a passing threshold value in the C-MCS$_{\text{out}}$ lock beyond which the HMCS$\langle 3 \rangle$ lock is always guaranteed to deliver higher throughput at the same fairness. This property also follows from the fact that $\mathcal{T}^{max}_{out}(3) < \mathcal{T}^{max}_{hmcs\langle 3 \rangle}(3)$.
2. When the first condition is not met, as long as the cost of additional check of the status flag and amortized cost of traversing to the parent level lock once in $h_1 = n_1$ quantum is less than three times the difference in passing time between level-1 and level-2, it is beneficial to have additional level in the lock hierarchy. The RHS is, typically, significantly less than the minimum possible LHS.

Discussion: It is straightforward to show that an N-level HMCS lock offers the same throughput and fairness superiority guarantees over an (N-1)-level HMCS lock. The argument follows similar to the aforementioned derivations; we omit the details for brevity.

	IBM Power 755	**SGU UV 1000**
Processor	POWER7 @ 3.86 GHz	Intel Xeon X7560 @ 2.27 GHz
SMT	4-way	2-way
Cores/Socket	8	8
Sockets/node	4	2
L1-D-Cache	32KB private	32KB private
L2-Cache	256KB private (L1 inclusive)	256KB private (L1 inclusive)
L3-Cache	8 × 4MB victim	24MB shared (L2 inclusive)
Memory controller	2 on-chip, 4-channel DDR3	2 on-chip, 2-channel DDR3
Compiler	`xlc++ v1.4.3`	`icc v.14.0.0`

Table 4.1 : Experimental setup.

4.8 Experimental evaluation of the HMCS lock

We conducted our experimental evaluation on two platforms—an IBM Power 755 and an SGI UV 1000. For all experiments in this section, we always bind the threads *densely.* Under dense thread binding, all SMT threads of a core are populated before populating the other cores; all cores of a socket are populated before populating another socket; all sockets of a node are populated before populating another node; and so on.

4.8.1 Evaluation on IBM Power 755

The IBM Power 755, a 3-level system, used for our experiments has the specifications shown in Table 4.1. This system provides a total of 128 hardware threads. SMTs sharing an L1 and L2 cache form level-1 of the NUMA hierarchy. All cores on the same socket form level-2 of the NUMA hierarchy. Four sockets sharing the primary memory form level-3 of the NUMA hierarchy.

4.8.1.1 Accuracy of analytical models

We inspected the compiler-generated assembly instructions appearing on the critical path and assigned costs (CPU cycles) to instructions. For memory access instructions, we assigned costs to instructions based on the access latency for different levels of the memory hierarchy. We relied on the POWER7 manual [93] and empirical measurement, among other resources [228] for determining the cost of instructions and access latencies.

For the HMCS lock, assigning costs is straightforward because of the deterministic be-

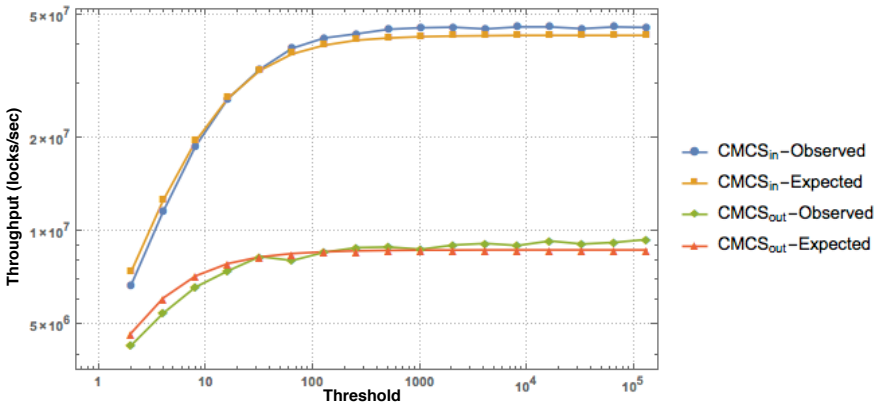


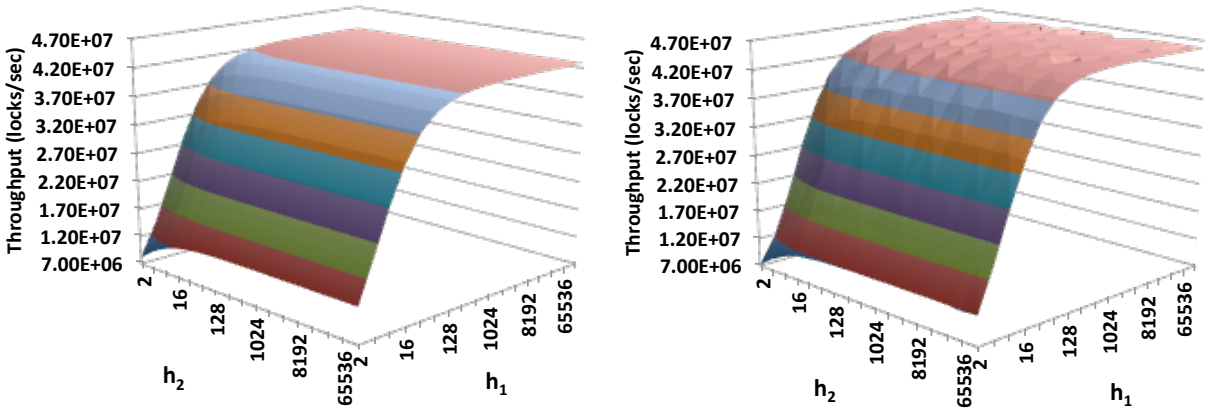
Figure 4.13 : Expected vs. observed throughput of the C-MCS$_{\text{in}}$ and C-MCS$_{\text{out}}$ locks on IBM Power 755 with 128 threads.

havior of the lock—h_1 number of local passes followed by one next level passing and so on, as discussed in the previous section. For the C-MCS locks, assigning costs follows the probabilistic model discussed in the previous section.

With the costs assigned to each instruction on the critical path, we computed the lock passing time for different levels—peer SMTs, peer cores, and peer sockets. By knowing the passing time at each level of the hierarchy, we computed the expected throughput values at various passing thresholds.

To assess the accuracy of our analytical models, we compared our model-derived throughput with the empirically observed throughput at 128 threads. Figure 4.13 plots the graph of expected vs. observed throughput of C-MCS locks. Figure 4.14(a) and 4.14(b) respectively plot the expected vs. empirically observed throughput of the HMCS$\langle 3 \rangle$ lock. The difference in expected vs. observed throughput is small (see Table 4.2). Clearly, our analytical models accurately predict the performance at each value of lock passing threshold. As expected, the peak throughputs of the HMCS$\langle 3 \rangle$ and C-MCS$_{\text{in}}$ locks are same ($4.58E{+}07$ acquisitions / second) and 4.8× higher than the C-MCS$_{\text{out}}$ lock.

Beyond a certain point, further increases in threshold yield no significant increase in the throughput of a lock. Significant performance gains happen in the HMCS$\langle 3 \rangle$ lock by increasing the h_1 threshold. Table 4.3 shows the difference in the unfairness of the C-MCS$_{\text{in}}$ lock and the HMCS$\langle 3 \rangle$ lock to reach a given target throughput. The HMCS$\langle 3 \rangle$ lock delivers



(a) Expected throughput

(b) Observed throughput

Figure 4.14 : Expected vs. observed throughput of HMCS⟨3⟩ lock on IBM Power 755 with 128 threads.

Lock	Median Difference	Maximum Difference
C-MCS$_{\text{in}}$	5.6%	10%
C-MCS$_{\text{out}}$	4.7%	11%
HMCS⟨3⟩	6.3%	15%

Table 4.2 : Difference in expected vs. observed throughput on IBM Power 755.

Percent peak throughput	Unfairness		(HMCS improvement)
	HMCS⟨3⟩	C-MCS$_{\text{in}}$	C-MCS$_{\text{in}}$ / HMCS⟨3⟩
50%	24	173	7.14×
70%	222	568	2.56×
90%	1209	2545	2.10×
99%	14543	29236	2.01×
99.9%	147880	296142	2.00×

Table 4.3 : Superior fairness of the HMCS⟨3⟩ lock over the C-MCS$_{\text{in}}$ lock.

up to 7x higher fairness than the C-MCS$_{\text{in}}$ lock. At 99.9% of the peak throughput, the HMCS⟨3⟩ lock delivers 2× higher fairness than the C-MCS$_{\text{in}}$ lock.

4.8.1.2 Empty critical section microbenchmark

Figure 4.15 demonstrates the scalability of various MCS lock variants with empty critical sections. We chose the passing thresholds to deliver 99.9% of the peak throughput for C-MCS and HMCS⟨3⟩ locks. Under no contention, the HMCS⟨3⟩ lock has 2.9× lower throughput than the MCS lock. Naturally, where there is no contention, the HMCS⟨3⟩ lock has 3×

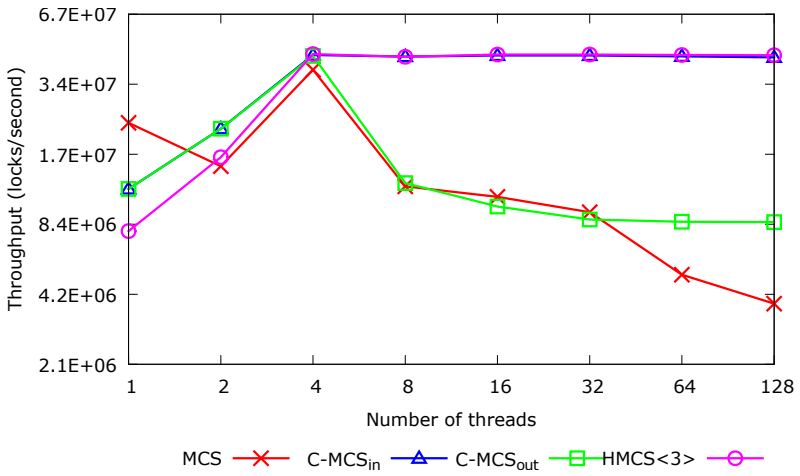


Figure 4.15 : Lock scaling with an empty critical section on IBM Power 755.

additional locking overhead. Under high contention, however, the HMCS⟨3⟩ lock has 11.7× higher throughput. The throughput of the MCS lock drops each time a new NUMA level is introduced. The C-MCS$_{in}$ lock follows a similar high-throughput trend as the HMCS⟨3⟩ lock. The throughput of the C-MCS$_{out}$ drops between 4-8 threads when the SMT-threads diverged into multiple cores. The HMCS⟨3⟩ lock delivers 5.22× higher throughput than the C-MCS$_{out}$ lock. At two processors, the MCS lock's throughput drops. This anomaly is explained in the original paper describing the MCS lock [157].

4.8.1.3 Non-empty critical section microbenchmark

We varied the data accessed in the critical section from 0 to 8MB while keeping the number of threads fixed at 128. Using our analytical models, we chose the passing thresholds that would deliver 99.9% of the peak throughput with empty critical sections. We compared the throughput of the HMCS⟨3⟩ lock to the C-MCS$_{out}$ lock (Figure 4.16). The throughput of the HMCS⟨3⟩ lock increases from 5.3× at 0 bytes to 7.6× at 32KB—the L1 cache size. The increase in relative throughput is because of the locality benefits enjoyed by the data accessed in the critical section. Beyond the L1-cache size, the ratio decreases, yet continues to be significantly superior (more than 6×) until 256KB—the L2 cache size. Beyond the L2-cache size, the benefits of the HMCS⟨3⟩ lock steadily drop from 6.2× to 1.5× at 4MB—the L3 cache size.

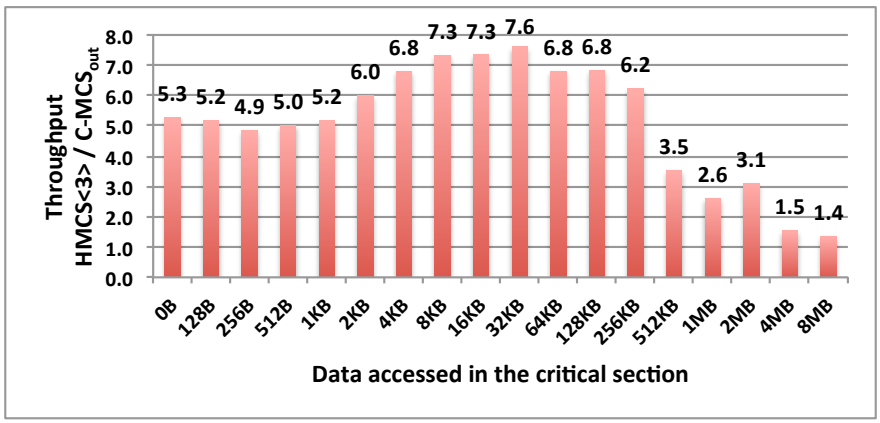


Figure 4.16 : Throughput improvement of HMCS⟨3⟩ over C-MCS$_{out}$ with varying size of data accessed in the critical section on IBM Power 755 with 128 threads.

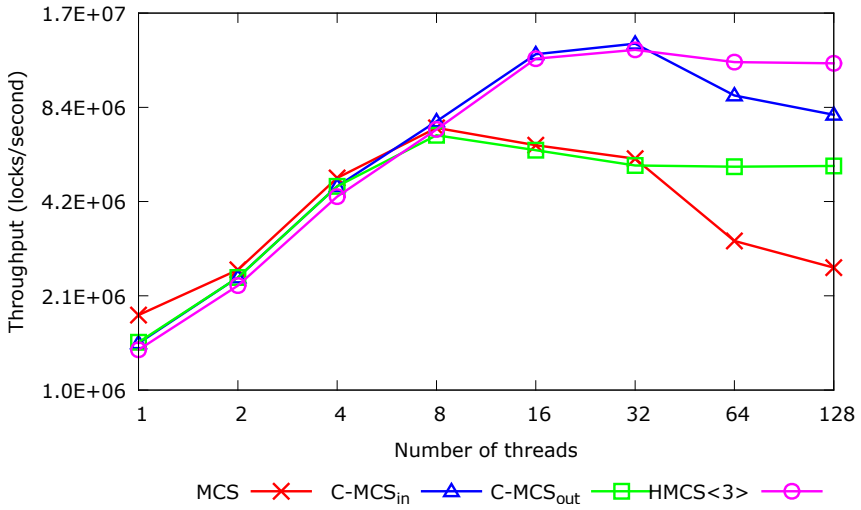


Figure 4.17 : Lock scaling at lower contention on IBM Power 755.

4.8.1.4 Non-empty critical section with lower contention

Figure 4.17 demonstrates the scalability of various MCS lock variants with non-empty critical sections under *lower contention*. Our benchmark touches two cache lines inside the critical section and spends a random 0-1.58 μs outside the critical section to mimic the test setup by Dice et al. [63]. We chose the passing thresholds to deliver 99.9% of the peak throughput for all cohort locks. In this case, the C-MCS$_{in}$ lock's throughput starts to degrade once the lock passing starts to go outside of the socket due to a lack of cohort formation. The HMCS⟨3⟩ lock maintains its high throughput and remains unaffected by increased NUMA

levels, whereas throughput of all other locks degrades. At 128 threads, the HMCS⟨3⟩ lock delivers 1.46×, 2.13×, and 4.5× higher throughput than C-MCS$_{\text{in}}$, C-MCS$_{\text{out}}$, and MCS locks respectively.

4.8.1.5 MineBench K-means code

K-means is an OpenMP clustering code from the MineBench v.3.0.1 suite—a benchmark suite with full-fledged implementations for data mining workloads [165]. "K-means represents a cluster by the mean value of all objects contained in it. The initial, user provided, k cluster centers are randomly chosen from the database. Then, each object is assigned a nearest cluster center based on a similarity function. Once the new assignments are completed, new centers are found by finding the mean of all the objects in each cluster. This process is repeated until some convergence criteria is met [165]."

Our experiments used an input file with 65K objects and 32 attributes in each point. We set the convergence threshold to 10^{-5}; we set the minimum and maximum number of initial clusters to 2 and 15, respectively. The K-means code uses OpenMP `atomic` directives to update the new clusters centers as shown in Listing 4.2. Data migrates indiscriminately among remote NUMA domains due to *true sharing* (threads in different NUMA domains modify the same location) and *false sharing* (threads in different NUMA domains modify different locations that share the same cache line). The result is poor scalability—increasing the number of threads increases the running time—as shown in Table 4.4 column #2.

To address the indiscriminate data movement, we replaced the atomic directives with a coarse-grained lock to protect the cluster centers (see Listing 4.3). While there exist other parallelization techniques that can deliver higher scalability, our single-lock solution serves to demonstrate the utility of the HMCS lock under high contention on a nontrivial use case. We employed the MCS, C-MCS$_{\text{in}}$, C-MCS$_{\text{out}}$, and HMCS⟨3⟩ locks as coarse-grained lock implementations. Respective running times are shown in columns #3-#6 in Table 4.4. For all cohort locks, we used a high passing threshold since fairness was immaterial. Columns #7-#10 show the improvements of each of the locks, at the same thread settings, when compared to the K-means that used atomic directives.

The NUMA-agnostic coarse-grained MCS lock improved K-means performance by up

```
1 /* update new cluster centers : sum of objects located within */
2 #pragma omp atomic
3 new_centers_len[index]++;
4 for (j=0; j<nfeatures; j++)
5 #pragma omp atomic
6   new_centers[index][j] += feature[i][j];
```

Listing 4.2: K-means atomic updates (poor performance).

```
1 /* update new cluster centers : sum of objects located within */
2 Acquire(lock, me);
3 new_centers_len[index]++;
4 for (j=0; j<nfeatures; j++)
5   new_centers[index][j] += feature[i][j];
6 Release(lock, me);
```

Listing 4.3: K-means coarse-grained locking (better performance).

col 1	col 2	col 3	col 4	col 5	col 6
	Running time in microseconds				
Num Threads	Atomic ops	MCS lock	C-MCS$_{in}$ lock	C-MCS$_{out}$ lock	HMCS⟨3⟩ lock
1	1.61E+08	4.34E+07	4.77E+07	4.78E+07	5.23E+07
2	1.29E+08	3.83E+07	3.70E+07	3.79E+07	4.03E+07
4	9.09E+07	2.95E+07	3.03E+07	3.03E+07	3.22E+07
8	1.28E+08	1.92E+07	1.82E+07	2.08E+07	1.90E+07
16	2.75E+08	2.58E+07	1.97E+07	2.67E+07	1.92E+07
32	4.02E+08	2.70E+07	1.77E+07	2.72E+07	1.76E+07
64	4.82E+08	5.62E+07	3.24E+07	3.56E+07	2.43E+07
128	6.49E+08	7.81E+07	5.67E+07	5.45E+07	3.93E+07

	col 7	col 8	col 9	col 10	col 11	col 12
	Improvement over atomic ops				HMCS⟨3⟩ improvement over	
Num Threads	MCS (#2/#3)	C-MCS$_{in}$ (#2/#4)	C-MCS$_{out}$ (#2/#5)	HMCS⟨3⟩ (#2/#6)	C-MCS$_{in}$ (#4/#6)	C-MCS$_{out}$ (#5/#6)
1	3.70×	3.37×	3.36×	3.07×	0.91×	0.91×
2	3.37×	3.49×	3.41×	3.21×	0.92×	0.94×
4	3.08×	3.01×	3.01×	2.83×	0.94×	0.94×
8	6.69×	7.06×	6.17×	6.75×	0.96×	1.09×
16	10.7×	14.0×	10.3×	14.4×	1.03×	1.39×
32	14.9×	22.7×	14.8×	22.9×	1.01×	1.55×
64	8.58×	14.9×	13.5×	19.8×	1.33×	1.46×
128	8.31×	11.4×	11.9×	16.5×	1.44×	1.39×

Table 4.4 : Comparison of different synchronization strategies for K-means on IBM Power 755.

to 14.9× over the atomic operations (Table 4.4, column #7). Even when there is no false sharing (1 thread), using a coarse-grained lock is superior to using multiple atomic add instructions, since each atomic add on POWER7 turns into a sequence of instructions that includes a *load reserve* and a *store conditional* resulting in higher instruction count. When contention rises, many threads simultaneously attempt to perform the load reserve and store conditional operations. One of them succeeds and many fail. The failed ones keep retrying choking the communication network.

The HMCS⟨3⟩ lock delivers 22.9× higher performance at 32 threads compared to the original execution that used atomic operations. This is 9.2× speedup compared to the original serial execution. Furthermore, beyond 16 threads, the HMCS⟨3⟩ lock's performance is better than all other locks for the same thread settings.

We compare the performance of the HMCS⟨3⟩ lock with the C-MCS$_{\text{in}}$, C-MCS$_{\text{out}}$ locks in columns #11-#12 respectively. The C-MCS$_{\text{in}}$ and HMCS⟨3⟩ locks start to show their superior performance at 8 threads since they have SMT-level lock passing. While the MCS and C-MCS$_{\text{out}}$ locks degrade in performance beyond 8-threads, the C-MCS$_{\text{in}}$ and HMCS⟨3⟩ locks continue to improve until 32 threads. At 32 threads, the C-MCS$_{\text{in}}$ lock degrades steeply since it does not pass locks among cores of the same socket. Due to lock passing at both SMT and core levels, the HMCS⟨3⟩ lock demonstrates superior performance over all other locks as the contention rises between 16-128 threads. The performance of the HMCS⟨3⟩ lock degrades at 64 and 128 threads compared to its own performance at 32 threads, which is due to the nature of the program itself. From column #12, we notice that the HMCS⟨3⟩ lock improves the performance by up to 1.55× compared to the C-MCS$_{\text{out}}$ lock. At lower contention, the HMCS⟨3⟩ lock incurs up to 9% slowdown compared to the two-level locks. This slowdown is expected due to the additional locking overhead of the third level.

4.8.2 Evaluation on SGI UV 1000

The SGI UV 1000 [208] used for our experiments consists of 256 blades. Each blade has two Intel Xeon X7560 processors, with the specifications shown in Table 4.1. Pairs of blades are connected with QuickPath interconnect. All nodes are connected via NUMAlink [208] technology. We had access to 4096 hardware threads out of the total 8192 on the system. We organized the SGI UV 1000 into a 5-level hierarchy as shown in Table 4.5.

For the C-MCS locks, we formed the cohorts at level-1 (C-MCS$_{\text{in}}$), level-2 (C-MCS$_{\text{mid}}$), and level-4 (C-MCS$_{\text{out}}$). We compared the C-MCS locks with a 5-level HMCS lock. In all locks, we set infinite threshold at each level but fixed number of iteration for each thread, so that the lock may not be needed again once relinquished to the parent. This configuration provided the highest possible throughput for any lock. We tested these locks both with empty and non-empty critical section. The non-empty critical section case touched two cache lines

Level	Participants	Total
1	SMTs	2
2	Cores	8
3	4 sockets (a pair of adjacent nodes) connected via QPI [97]	64
4	8-pairs of nodes on the same rack	512
5	8 racks	4096

Table 4.5 : NUMA hierarchy of SGI UV 1000 exploited by HMCS lock

	Throughput mode			
	HMCS$\langle 5 \rangle$	C-MCS$_{\text{in}}$	C-MCS$_{\text{mid}}$	C-MCS$_{\text{out}}$
Empty critical section	3.09E+07	1.26E+06	9.76E+06	4.32E+05
(HMCS improvement)	-	(24.6x)	(3.17x)	(71.5x)
Non-empty critical section	5.63E+06	5.27E+05	4.34E+06	2.61E+05
(HMCS improvement)	-	(10.7x)	(1.3x)	(21.5x)

Table 4.6 : Throughput (locks/sec) improvement of HMCS vs. C-MCS locks on a 4096-thread SGI UV 1000.

	Fairness mode			
	HMCS$\langle 4 \rangle$	C-MCS$_{\text{in}}$	C-MCS$_{\text{mid}}$	C-MCS$_{\text{out}}$
Empty critical section	2.43E+06	5.02E+05	2.28E+06	4.11E+05
(HMCS improvement)	-	(4.84x)	(1.07x)	(5.92x)
Non-empty critical section	5.63E+06	3.78E+05	1.71E+06	3.96E+05
(HMCS improvement)	-	(14.9x)	(3.29x)	(14.2x)

Table 4.7 : Throughput (locks/sec) improvement of HMCS vs. C-MCS locks on a 4096-thread SGI UV 1000.

inside the critical section and spent 0-2.5 microseconds outside the critical section to mimic the test setup by Dice et al. [63]. Table 4.6 shows that the HMCS lock outperforms the peak throughput of all other locks in both modes. The HMCS lock outperforms even the C-MCS$_{\text{in}}$ because the threshold c_{in} needed to amortize the latency of deep NUMA hierarchy is larger than the number of lock acquisitions needed by each thread.

We also compared the C-MCS locks with a 4-level HMCS lock, where the level-1 and level-2 were collapsed into a single domain. In this case, we set the threshold at each level to the number of participants at that level. This configuration provided the highest fairness for any lock. As before, we tested these locks with both empty critical sections and non-empty critical sections. Table 4.7 shows that the HMCS lock outperforms the peak throughput of all other locks in both modes.

Finally, we evaluated the scalability of the MCS, C-MCS$_{\text{mid}}$, and HMCS$\langle 5 \rangle$ locks on SGI

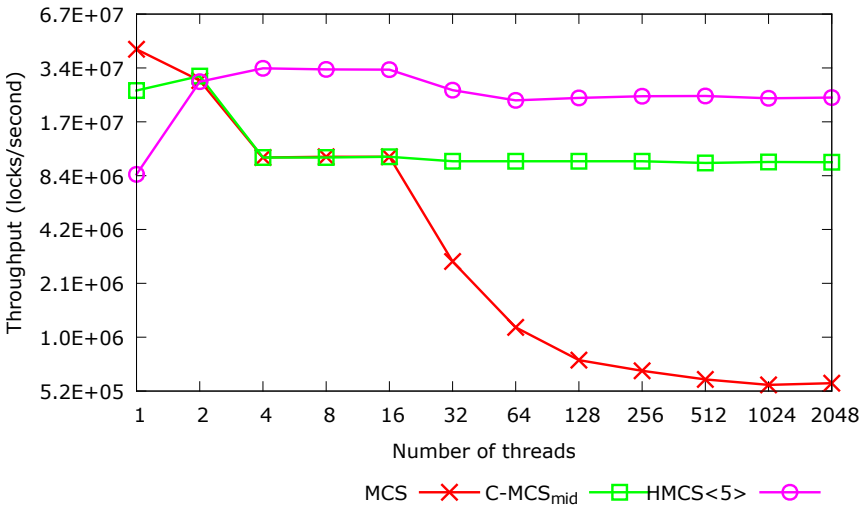


Figure 4.18 : Lock scaling with empty critical sections on SGI UV 1000.

UV 1000. We restricted our scaling studies to just these three locks due to allocation time limitations on the machine. We again used empty critical sections and no delay outside the critical section. Figure 4.18 plots the scalability of the three locks. Clearly, the MCS lock has low scalability. Remember, the threads are densely packed. Throughput drops particularly when a new NUMA domain is introduced, for example at x=4 (passing to a different core), x=32 (passing to a different sockets), x=64 (passing to a node 1 hop away), x=128 (passing to a node 2 hops away), etc. The C-MCS$_{\text{mid}}$, which is a 2-level lock with cohorts being formed among the cores sharing the same socket, drops in performance when the passing is not within the SMT threads. C-MCS$_{\text{mid}}$, however, maintains a high throughput remaining largely unaffected by locality losses arising from off-socket passing. Finally, the HMCS$\langle 5 \rangle$ lock has the highest scaling and maintains a steady throughput, which is much higher than both MCS lock and C-MCS$_{\text{mid}}$ lock. The HMCS$\langle 5 \rangle$ lock is slightly affected by locality losses arising from off-socket and off-node passing.

4.9 Adaptive HMCS locks[2]

Applications may exhibit different levels of lock contentions during different phases execution. Also, different parts of the same system may exhibit different contention levels at the same time. A fixed-depth HMCS lock incurs the overhead of additional lock acquisitions that is unnecessary under low contention. A hierarchical lock needs to address the following cases:

Zero contention: When there is no contention in the entire system, the *latency* should be low. Ideally, the latency should match the uncontended acquisition in a flat queuing lock, such as MCS lock (referred to as HMCS$\langle 1 \rangle$ hereafter).

Full contention: When the contention is very high in the entire system, the *throughput* should be high. Ideally, the throughput should match the peak throughput of a fully contended acquisition in an HMCS lock of the deepest depth appropriate for a given system.

Moderate contention: When the contention in the system is neither too low to justify an MCS lock nor too high to use a deep HMCS lock, the locking algorithm should balance latency and throughput.

We begin with an HMCS$\langle n \rangle$ lock,[3] so that the case of full contention is already addressed. To reduce the latency of acquisitions under *zero* contention, we augment the HMCS lock with a fast-path mechanism. The fast-path mechanism, discussed in Section 4.9.1, bypasses a chain of lock acquisitions in an HMCS tree when a thread requests the lock without contention from any other thread. We address *moderate* contention with a hysteresis approach. The hysteresis approach for *moderate* contention is discussed in Section 4.9.2. Finally, combining an HMCS$\langle n \rangle$ lock with the fast-path and hysteresis mechanisms addresses all levels of contention.

---

[2]The adaptive HMCS locks were developed in response to a probing question about the cost of uncontended lock acquisitions in HMCS by Prof. Michael Scott (University of Rochester) at PPoPP 2015.

[3]n will be chosen to be profitable on a target system's NUMA hierarchy.

Terminology: We consider a leaf of the tree in an HMCS locking hierarchy to be at *level* 1 and the root to be at *level* n. Conversely, we consider a leaf to be at *depth* n and the root to be at *depth* 1. We use the terms *depth* and *level* as appropriate in the rest of this chapter. In the rest of this chapter, we represent an arbitrary level in an HMCS tree with the letter h, which signifies the height starting from the leaves. Naturally, $h-1$ will be closer to leaves, and $h+1$ will be closer to the root.

4.9.1 Making uncontended acquisitions fast with a fast-path

When there is no contention in the system, a deep HMCS$\langle n \rangle$ lock incurs the cost of n lock acquisitions (and releases). This overhead may be undesirable if the lock acquisition is frequent but acquisitions are often uncontended.

To reduce the latency of uncontended acquisition, we enhance the HMCS lock with a fast-path mechanism (FP-HMCS). Figure 4.19 depicts the FP-HMCS lock, and Algorithm 3 provides the pseudocode. FP-HMCS's acquire protocol checks if the tail pointer of the root-level MCS locks is null. If it is null, then the protocol infers the lock is uncontended and directly enqueues the `QNode` at the root level (*fast-path*); otherwise, it follows the normal HMCS protocol via leaf of the tree (*slow-path*). The fast-path allows the uncontended acquisition to bypass acquiring several lower level locks. If a thread is the first one to enqueue at the root level (common when uncontended), then it acquires the lock immediately; otherwise, it waits (uncommon when uncontended) until its predecessor passes the lock to it, possibly after passing within its subdomain. A thread that decides to enqueue directly at the root level sets a thread-local flag `tlTookFastPath` to `true`. FP-HMCS's release protocol invokes the root-level release protocol if the `tlTookFastPath` flag is set, unsetting it subsequently.

A key design point of the HMCS lock was to eliminate accessing shared lock data structures that may be present in remote caches on NUMA machines. The aforementioned querying of the tail pointer of the root-level lock may seem to violate this principle. The following three properties hold when the system is contended, ensuring that we do not violate the locality principle.

1. The tail pointer of the root-level node will be non-null. This property deflects all new

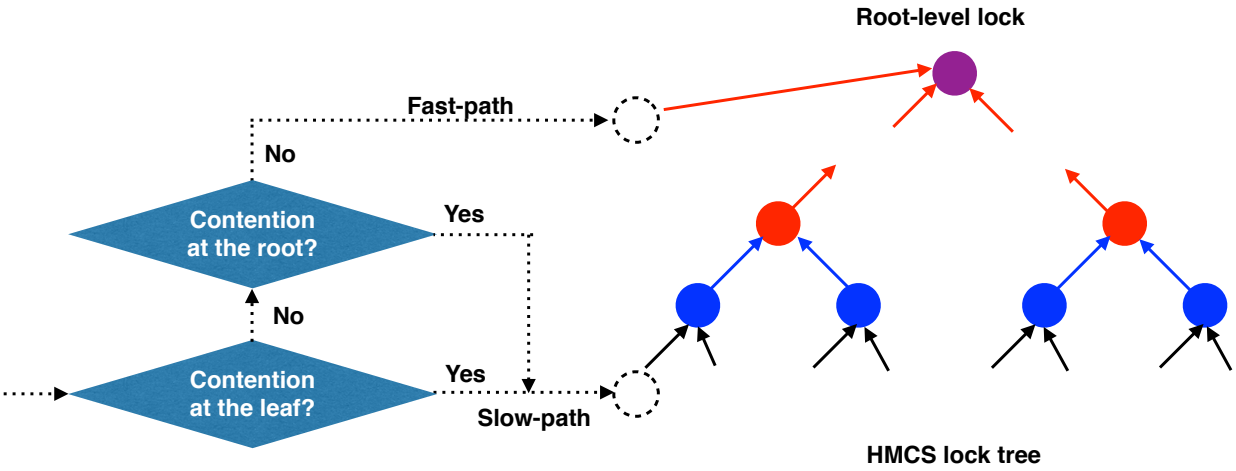


Figure 4.19 : Fast-path decision in the FP-HMCS lock. If the root-level lock is not taken, the FP-HMCS lock takes the fast-path and directly starts the root level (HMCS$\langle 1 \rangle$) acquisition protocol; otherwise it follows the slow-path through the HMCS$\langle n \rangle$ protocol.

Algorithm 3: Fast-Path HMCS (FP-HMCS) algorithm

```
1  ThreadLocal boolean tlTookFastPath /* Initially false                    */
2  void Acquire(...)
3      if IsLeafLevelLockFree() and IsRootLevelLockFree() then
4          tlTookFastPath ⟵ true
5          Perform HMCS root-level Acquire
6      else
7          Perform HMCS leaf-level Acquire
8  void Release(...)
9      if tlTookFastPath then
10         Perform HMCS root-level Release
11         tlTookFastPath ⟵ false
12     else
13         Perform HMCS leaf-level Release
```

acquisitions to their HMCS analogs, causing them to enqueue in their local domains (at HMCS tree leaves).

2. The tail pointer of the root-level lock does not change rapidly. This sluggishness is because new lock requests enqueue in their local domains when the lock is contended, and each level performs local passing. A thread, wanting to acquire the lock, seldom climbs to the root of the tree, especially when the passing threshold is high.

3. A recent value of the root-level tail pointer is usually present in the nearest cache of each thread. This property holds true since 1) a peer thread would have recently

checked the root-level tail pointer, and 2) the root-level tail pointer does not change frequently.

To avoid unnecessarily querying the root-level tail pointer on each acquisition, FP-HMCS first checks if the tail pointer of the leaf-level[4] lock is null. A non-null leaf-level tail pointer indicates contention; consequently, FP-HMCS takes the *slow-path* through the HMCS tree without checking the root-level lock. Once a local lock's tail pointer is non-null, other threads arriving in the same domain do not check the root-level lock. This small modification along with the properties of the HMCS lock under contention ensures the efficiency of the fast-path scheme. *The fast-path mechanism is beneficial when the system is uncontended and inexpensive when the system is contended.*

The empirical studies later in Section 4.11 show that under zero contention, an FP-HMCS$\langle 3 \rangle$ remains within 8% of the performance of an MCS on a Power 755. In addition, under full contention an FP-HMCS$\langle 3 \rangle$ remains within 8% of the performance of an HMCS$\langle 3 \rangle$ on a Power 755.

We note that our fast-path mechanism is analogous to the one in Yang and Anderson's lock [240]. A key difference from Yang and Anderson's lock is that they introduce an additional 2-thread mutual exclusion atop their binary arbitration tree, both on slow- and fast-paths. Contrary to Yang and Anderson's fast-path mechanism, our technique does *not* require any additional lock levels to support the fast-path. The queuing property of the MCS lock allows multiple threads to take the fast-path simultaneously and get enqueued at the root level.

4.9.2 Adapting to various contention levels using hysteresis

The aforementioned fast-path technique can solve only the overhead of uncontended lock acquisitions. If contention exists but it is not high enough to take advantage of local passing at the leaf-level domains of an HMCS tree, the tree needs to "right-sized" such that it is deep enough to exploit local passing but shallow enough to avoid any unnecessary locking overhead.

[4]We later alter this to check the "current" level based on the hysteresis, instead of the leaf level alone.

To balance latency and throughput under a variety of contention levels, we modify the HMCS lock into an **A**daptive **H**ierarchical **MCS** (AHMCS) lock. *Instead of modifying the structure of the tree, which is complicated, we modify the protocol such that the threads may begin their acquisition (and release) protocols at any interior node in the tree. The key idea is to use the previously observed contention as a predictor to target a level in the HMCS locking hierarchy to begin the next round of acquisition.* This adaptation allows threads to dynamically skip lower levels of the HMCS tree that may not be sufficiently contended.

An n-level AHMCS lock is a wrapper around an n-level HMCS lock. Recall that each thread in an HMCS lock brings its `QNode` and enqueues it at the leaf of the HMCS tree. Also recall that each thread uses a preallocated designated `QNode` when enqueueing at an interior node. Further, recall that each acquisition begins at the leaf of the tree in the HMCS protocol. The AHMCS lock deviates from this behavior in the following ways:

1. Each thread in an AHMCS may skip some lower $h - 1$ levels of the tree to begin the acquire and release protocols directly at level h.

2. A thread that begins the acquisition protocol at level h, enqueues its `QNode` at level h and does *not* use the `QNode` designated for its domain at that level. This protocol ensures no race between a thread that would have progressed from a lower level m (where $m < h$) to the level h and threads enqueuing directly at an interior level h. The effect is analogous to adding an additional peer domain in the HMCS tree, but such "ghost" domains will have no subdomains.

3. The user code no longer provides a `QNode`. The user code maintains a handle to the AHMCS lock. The AHMCS lock object pointed to by the handle wraps a `QNode`.

We discuss how each thread tracks contention in its domain and determines which level to enqueue in the HMCS tree in the following subsections. Depending on the contention in a domain, within a domain (say D), threads may skip the lower h levels of the HMCS tree, whereas at the same time, within another domain (say E), the threads may skip only the lower q (where $q < h$) levels of the HMCS tree. A thread that begins its acquire protocol at a level h, also begins its release protocol at the same level.

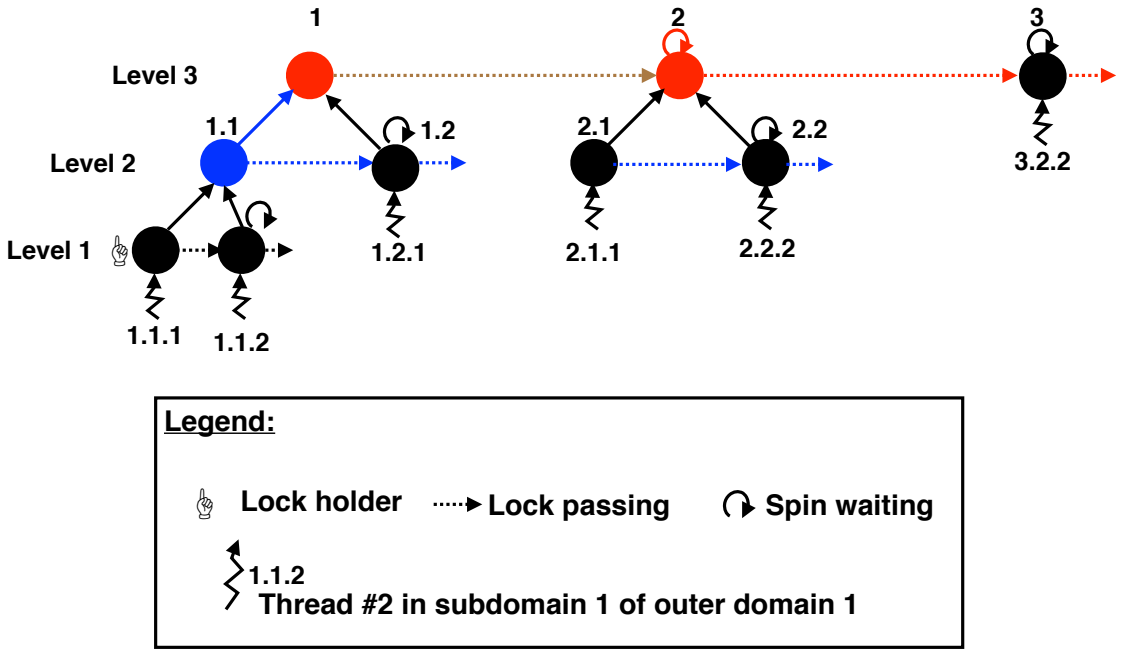


Figure 4.20 : A snapshot view of the Adaptive HMCS (AHMCS) lock. Different threads enqueue at different levels depending on the contention. For example, thread 1.1.1 and 1.1.2 create enough contention and enqueue at the lowest level (leaf) to take advantage of locality. The thread 3.2.2 has no contention at level 1 and level 2, as a result, it directly enqueues at level 3 to reduce latency.

Initially, when no knowledge of contention is available, each thread begins its acquire protocol by enqueuing at the leaf level of the HMCS tree. As contention levels change, threads move up and down, enqueuing at different levels where the contention is appropriate for exploiting locality with threads from peer domains.

Figure 4.20 shows an example snapshot of a 3-level AHMCS lock. In the figure, the threads 1.1.1 and 1.1.2 create sufficient contention inside the subdomain 1.1; as a result, these two threads enqueue at the leaf of the HMCS tree (level 1). The domain 1.2 has only one participant, thread 1.2.1. Hence, it does not enqueue at level 1. The subdomains 1.1 and 1.2 have enough contention to take advantage of the locality; hence, the thread 1.2.1 enqueues at level 2. Similarly, threads 2.1.1 and 2.2.2 each have no peers at level 1 but together they create enough contention at level 2, and hence they enqueue at level 2. Finally, the thread 3.2.2 observes no contention at level 1 and level 2 in its subdomains, as a result, it directly enqueues at level 3.

Adjusting to changing contention with hysteresis: The AHMCS lock uses *hysteresis* to predict the future contention based on the past and current contention. Initially, a thread T begins its AHMCS lock acquire protocol at the leaf level. Eventually, by observing the past contention, the AHMCS lock decides the appropriate level to start its enqueue process in the HMCS tree. This adaptation involves either moving up in the tree closer to the root when contention decreases or moving down closer to leaves in the tree when contention rises.

Each thread can infer if it is benefiting from local passing by identifying if the lock at level h, where it started its acquisition process, was acquired from its predecessor or passed to a successor. Similarly, a thread can infer low contention at a level based on the lack of a predecessor or a successor.[5]

We offer two solutions to adjust to the changing contention. One approach is *eager*. The eager approach is aggressive in changing its level on observing a change in contention. Another approach is *lazy*. The lazy approach is conservative; hence, it observes contention for several rounds of acquisition before deciding to change a level. We present the details of both eager and lazy strategies in Section 4.9.2.1 and 4.9.2.2, respectively. When we want to refer to an adaptive HMCS lock in a generic manner, without any distinction between an eager or a lazy implementations, we use the term "AHMCS".

4.9.2.1 Eager adaptive HMCS lock

The eager approach is aggressive in changing levels when it observes any change in contention. It uses the contention observed only in the previous round of lock acquisition in conjunction with the contention observable now to decide where to begin the current acquisition process. The eager approach has an affinity to begin acquisition at a level closer to leaves to exploit locality.

Intuitively, if a thread observes no contention at its current level, it targets one level closer to the root in the immediate next round of acquisition; if a thread observes contention at its current level, it targets one level closer to its leaf in the immediate next round of acquisition.

[5]One can explore several combinations such as conjunction or disjunction on the presence of a predecessor and a successor to infer contention. One can also choose an optimistic approach by considering only the presence of a successor and neglect a predecessor to infer contention.

The eager AHMCS lock exhibits its eagerness in yet another way; if the target level becomes contended just before starting the next round of acquisition, the protocol instantaneously backs off a level closer to its leaf and begins the acquisition process at that level. In fact, this instantaneous back-off is the premise of how the protocol understands the rise in contention after a phase of ebb.

Two terms need to be distinguished: *target-level* is where the protocol *intends* to begin the acquisition process, and *start-level* is where the protocol *actually* begins the acquisition process after its instantaneous decisions. *Target-* and *start-* levels may differ by one level.

Rising towards root on contention reduction: If the *target-level* h in this round was uncontended during acquire phase (had no predecessor node) and release phase (had no successor), the thread chooses level $h + 1$, which is closer to the root, as its *target-level* for the immediate next round.

Receding towards leaves on contention increase: It is non-trivial to recognize an increase in contention arising from a thread `T`'s subdomain, after a phase of contention reduction. The complication arises because the queue formed at a target level h will be intermixed with nodes enqueued by threads from the same subdomain as `T` as well as peer subdomains at level h.

The instantaneous decision to start a level lower than the target level, when the target level lock is not held, serves to solve this problem. More formally, a thread `T` targeting a level h makes the following *instantaneous* decisions:

1. If the lock at its *target-level* h is already held (the tail pointer of the lock at level h is non-null), it begins its acquire protocol at level $h - 1$; that is, *start-level* will be $h - 1$.

2. If the lock at its *target-level* h is not held (the tail pointer of the lock at level h is null), it begins its acquire protocol at level h; that is, *start-level* will be h.

When the lock at a target level h is already held, enqueuing at level $h - 1$, typically, incurs no overhead even if there is no local passing possible at level $h - 1$. This property is true because, the thread `T` would have, typically, waited at level h even if it had directly begun its acquisition at that level. Eagerly enqueuing at level $h - 1$ has two advantages:

Algorithm 4: EH-AHMCS algorithm

```
1  ThreadLocal int tlTargetLevel /* Initially n                                   */
2  ThreadLocal int tlStartLevel
3  ThreadLocal int tlLowestPredLevel
4  void Acquire(...)
5      if IsLockFree(tlTargetLevel) or
6         HaveNoChild(tlTargetLevel) or
7         IsLockHolderSameAsLastSuccessor(tlTargetLevel) then
8          tlStartLevel ⟵ tlTargetLevel
9      else
10         tlStartLevel ⟵ tlTargetLevel -1
11     tlLowestPredLevel ⟵ Perform HMCS Acquire at tlStartLevel   /* returns the lowest level where
                                                                     it found a predecessor */
12 void Release(...)
13     lowestSuccLevel ⟵ Perform HMCS Release at tlStartLevel     /* returns the lowest level where
                                                                     it found a successor */
14     if HaveChildLevel(tlTargetLevel) and
15        (tlLowestPredLevel < tlTargetLevel or lowestSuccLevel < tlTargetLevel) then
16         tlTargetLevel ⟵ tlTargetLevel -1
17         return
18     if HaveParentLevel() and
19        (tlLowestPredLevel > tlTargetLevel and lowestSuccLevel > tlTargetLevel) then
20         tlTargetLevel ⟵ tlTargetLevel + 1
21         return
```

1. It helps a thread to identify any rise in contention from its subdomain since a) the other threads from its subdomain that target level h will also back off by a level to start at level $h-1$, and b) if some threads were already at a level lower than h, at least one representative thread would rise till level $h-1$ in its acquisition process making contention at level $h-1$ evident to other threads that start at that level,

2. If the contention within `T`'s subdomain increases (and hence the level h is incorrect), starting at level $h-1$ instantaneously exploits the benefits of local passing at level $h-1$, instead of having to wait for the next round.

If the contention remains high or increases, eventually each thread progressively recedes to some level m closer to leaves ($m < h$), where local passing is beneficial. If the *start-level* h in this round was contended during acquire phase (had a predecessor node) and release phase (had a successor node), in the immediate next round, `T` chooses level $h-1$, which is closer to leaves, as its *target-level*.

Algorithm 4 presents the Eager Adaptive HMCS acquire and release protocols. Note that this algorithm does not have the aforementioned fast-path. We call this lock variant

as EH-AHMCS (**E**ager **H**ysteresis **A**daptive **H**ierarchical **MCS**) lock. Note also that this algorithm assumes that the underlying HMCS returns the level where the waiting and passing happened during the acquire and release phases respectively.

Protocol summary: Let, h be the level where a thread arrived based on the previous acquisition. The thread will begin the acquisition process at a level $p = [h, h-1]$. During the acquire protocol, the thread observes the level l, where it first spin waited for a lock. Similarly, in the release protocol, the thread observes the level m, where it passed a lock to a peer. If $l < h$ and $m < h$, the thread is benefiting from local passing at a level less than h. Consequently, the thread `T` arrives closer to its leaves, at level $h-1$, on the next round of acquisition. If $l > h$ and $m > h$, the thread is not benefiting from local passing at level h. Consequently, the thread `T` arrives closer to the root, at level $h+1$, on the next round of acquisition. Otherwise, the thread `T` continues to arrive at level h on the next round of acquisition.

Anecdotal examples: Figure 4.22 shows an example, where a thread T_1 targets level h and notices no contention at level h. T_1 enqueues its node X at level h (start-level), subsequently climbing to level $h+1$ to enqueue the node R—the representative node for its domain. T_1 does not see any contention at level h on release either. In the next round of acquisition, hysteresis suggests T_1 to target level $h+1$, avoiding the latency of acquiring the level h lock.

Figure 4.23 shows an example, where a thread T_1 targets level h and notices the presence of another node A at level h. T_1 enqueues its node X at level $h-1$ (start-level), subsequently climbing to level h to enqueue the node D—the representative node for its domain. Level $h-1$ becomes contended (thread S enqueues after X). Eventually, T_1 passes the lock to its successor S at level $h-1$. In the next round of acquisition, hysteresis suggests T_1 to target level $h-1$ (target level), where it may benefit by local passing from a predecessor.

Figure 4.24 shows an example, where a thread T_1 targets level h and notices the presence of another node A at level h. T_1 enqueues its node X at level $h-1$ (start-level), subsequently climbing to level h to enqueue the node D—the representative node for its domain. During

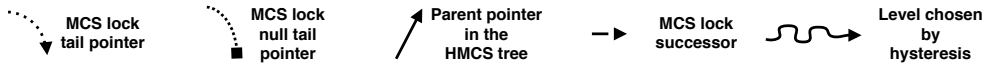
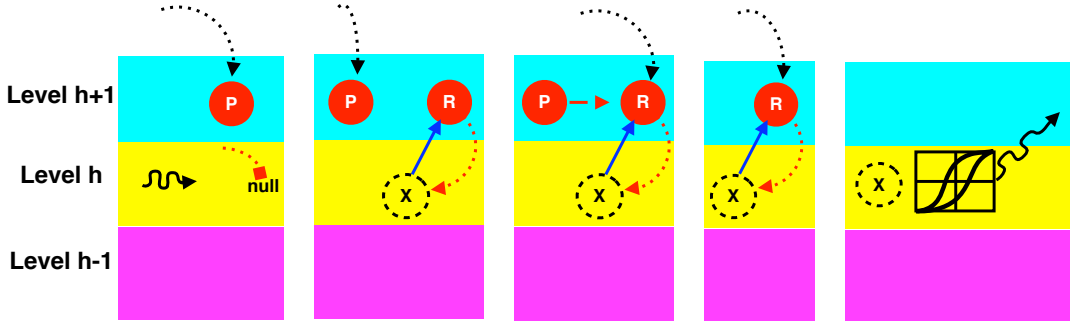
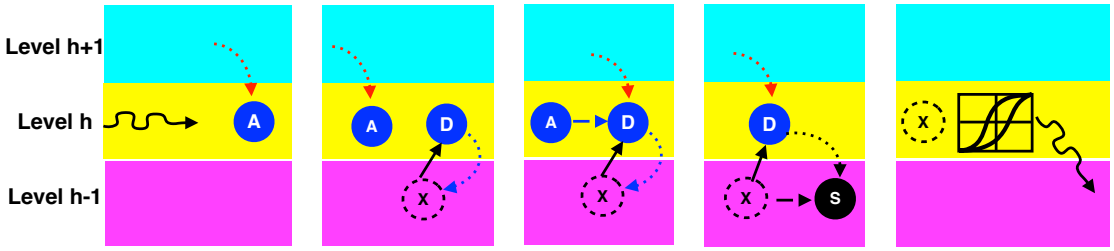


Figure 4.21 : Legend for Figures 4.22-4.24.



(a) Initial state (b) Acq. step 1 (c) Acq. step 2 (d) Release (e) Next round

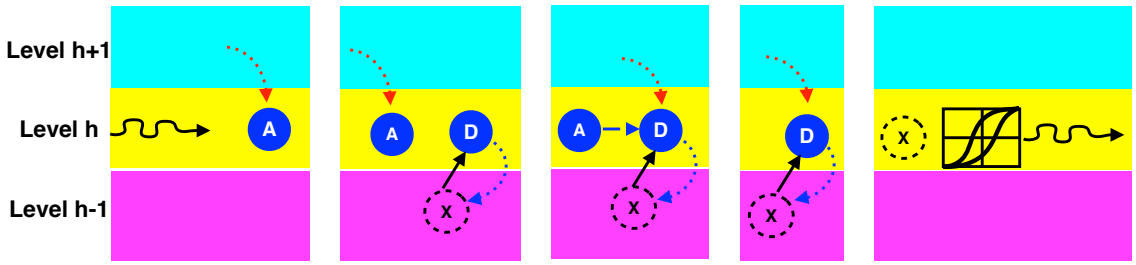
Figure 4.22 : Figure 4.22(a) represents the initial state, where no thread is enqueued at level h and level $h+1$ has an already enqueued node P. A thread (say T_1) brings a node X and wants to begin its acquire protocol at level h. T_1 checks the tail pointer at level h and recognizes no contention. The EH-AHMCS lock enqueues T_1's node X at level h, as shown in Figure 4.22(b). T_1 climbs to level $h+1$ and enqueues its representative node R at that level, as shown in Figure 4.22(c). During T_1's release, level h remains uncontended, as shown in Figure 4.22(d). Hysteresis suggests T_1 to target level $h+1$ during the next round of acquisition, as shown in Figure 4.22(e).



(a) Initial state (b) Acq. step 1 (c) Acq. step 2 (d) Release (e) Next round

Figure 4.23 : Figure 4.23(a) represents the initial state, where a node A is already enqueued at level h and a thread (say T_1) brings a node X and wants to begin its acquire protocol at level h. T_1 checks the tail pointer at level h and recognizes contention. The EH-AHMCS lock enqueues T_1's node X at level $h-1$, which is uncontended, as shown in Figure 4.23(b). T_1 climbs to level h and enqueues its representative node D at that level, as shown in Figure 4.23(c). During T_1's release, level $h-1$ becomes contended with a successor S, as shown in Figure 4.23(d). Hysteresis suggests T_1 to target level $h-1$ during the next round of acquisition, as shown in Figure 4.23(e).

its release, T_1 does not see any successor at level $h-1$. In the next round of acquisition, the hysteresis suggests T_1 to target level h (target level).



(a) Initial state (b) Acq. step 1 (c) Acq. step 2 (d) Release (e) Next round

Figure 4.24 : Figure 4.24(a) represents the initial state, where a node A is already enqueued at level h and a thread (say T_1) brings a node X and wants to begin its acquire protocol at level h. T_1 checks the tail pointer at level h and recognizes contention. The EH-AHMCS lock enqueues T_1's node X at level $h-1$, which is uncontended, as shown in Figure 4.24(b). T_1 climbs to level h and enqueues its representative node D at that level, as shown in Figure 4.24(c). During T_1's release, level $h-1$ remains uncontended, as shown in Figure 4.24(d). Hysteresis suggests T_1 to target level h during the next round of acquisition, as shown in Figure 4.24(e).

Handling a special case: Since the EH-AHMCS lock aggressively decides to enqueue a node at level $h-1$ on observing a non-null tail pointer at level h, it may take a performance penalty if there are not sufficient number of contenders at level h. A particular example is where a level h has only two threads ($\texttt{T}_1$ and $\texttt{T}_2$) and the critical sections are short. Assume a scenario where $\texttt{T}_1$ is enqueued at level h and just about to enter its critical section. If thread $\texttt{T}_2$ had enqueued behind $\texttt{T}_1$, then they could have taken the advantage of local passing. However, since the thread $\texttt{T}_2$ notices the tail pointer at level h to be non-null, it first enqueues at level $h-1$, where it finds no predecessor and then eventually enqueues a node at level h. When the critical sections are short, in this small window, $\texttt{T}_1$ might relinquish the lock to level $h+1$, instead of passing to $\texttt{T}_2$. Even when both $\texttt{T}_1$ and $\texttt{T}_2$ identify level h to be the correct level for them to enqueue, the eagerness of enqueueing a node at level $h-1$ results in a missed opportunity for local passing. In fact, such pattern leads to $\texttt{T}_1$ and $\texttt{T}_2$ incurring the lock passing latency at level $h+1$. To address this one case, we make a minor modification to the EH-AHMCS lock algorithm. A thread memorizes its successor S when it passes the lock to a peer.[6] If, on the immediate next arrival, the tail pointer continues to point to S, it means the contention is insufficient to justify a detour via

[6]No special memory is needed since `QNode`'s `next` field automatically contains a pointer to its successor.

level $h-1$. When a thread observes this situation, it continues to enqueue at level h and avoids eagerly enqueuing at level $h-1$. We have empirically noticed that this optimization is valuable occasionally where there are only two contenders for an interior node.

Unit stride vs. leap jump: A thread that targets level h, but both acquires the lock and passes the lock at a level l closer to the root ($l > h$), can easily recognize that level l is appropriate to target on the next occasion. This indication can help us avoid the unit-step progression from level $h+1$ to l. With this technique, adjusting to a reduction in contention can happen in a single round of acquisition instead of $d-1$ rounds of acquisitions for a tree of depth d. Similarly, when a thread notices that level h is contended, instead of starting its acquisition at level $h-1$, it can look for the level m closest to its leaves ($m < h$), where the tail pointer is non-null. Subsequently, if the thread passes the lock at level m, it can infer that the level m is appropriate for it to arrive on the next occasion. With this technique, if the contention suddenly rises, a thread `T` will be pushed down from the root to its leaf in just $d-1$ rounds of lock acquisitions. All $d-1$ acquisitions need not come from the thread `T` itself. Acquisitions originating from the threads sharing parts of the spine of the tree from the root to the leaf contribute to the $d-1$ acquisitions needed to push `T` down to its leaf level. Our implementation of the EH-AHMCS lock does not capture this leap jumping of levels. We leave this for our future work.

4.9.2.2 Lazy adaptive HMCS lock

The lazy approach is opposite to the eager approach; it resists changing levels as much as possible unless it has gathered enough evidence to benefit by changing a level. To gather sufficient evidence, the lazy approach observes contention for several rounds of acquisition before recommending a change. The lazy approach is useful in suppressing sporadic changes in contention. Note that this algorithm does not have the aforementioned fast-path. We call this lock variant as LH-AHMCS (**L**azy **H**ysteresis **A**daptive **H**ierarchical **MCS**) lock.

Receding towards leaves on contention increase: The lazy approach uses the same idea as the eager approach in deciding to rise up in the tree when it does not see contention

at a given target level. However, it does so cautiously. If a thread notices no contention for N successive times at its target level h, where N is a tunable parameter, then on $(N+1)^{th}$ round of acquisition, the thread begins to target level $h+1$, which is closer to root.

Receding towards leaves on contention increase: Similar to an EH-AHMCS lock, even in LH-AHMCS lock, it is non-trivial to recognize an increase in contention arising from a thread `T`'s subdomain, after a phase of contention reduction. The complication arises because the queue formed at a target level h will be intermixed with nodes enqueued by threads from the same subdomain as `T` as well as peer subdomains at level h.

In the lazy approach, when a thread enqueues at a higher level in the hierarchy, it advertises its presence the other threads from its lower subdomains. We use a shared counter to achieve this. Since threads from the same domain share an `HNode`, they can express their presence to one another by incrementing a counter placed in the designated `HNode` for each domain as they perform an acquire and a release. We use a 64-bit `contentionCounter` in each `HNode` in our implementation. A thread that enqueues at a level h, atomically increments and remembers the value of the `contentionCounter` designated for its domain.[7] During the lock release, if the value of the `contentionCounter` is different from the value recorded at the beginning of the acquire protocol, then there is at least one more thread from the same subdomain now present at level h. If a thread notices that its designated `contentionCounter` at its target level is different from an acquire to the corresponding release on N successive rounds of acquisition, it infers that there is enough contention to take advantage of locality within its subdomains and starts to target closer to its leaves at level $(h-1)$ on $(N+1)^{th}$ round of acquisition.

Even after introducing the contention counter, the following two subtle cases arise:

1. Only one thread has progressed to level $h+1$ but all other threads from the same subdomain continue to target level h. In this case, the solo thread that has progressed to level $h+1$ will be blind to the contention at level h since the `contentionCounter` will be same from acquisition to release.

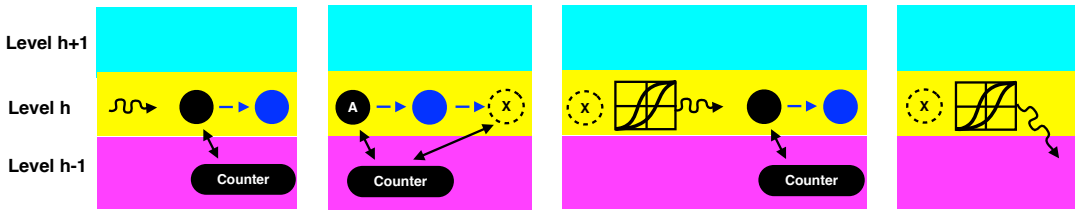
[7] Obviously, two threads that belong to two peer domains at level h update two different `contentionCounter`s.

2. Only one thread has remained at level h and all other threads target level $h+1$. In this case, the solo thread that enqueues at level h will be blind to the presence of other threads from the same domain since it would acquire without contention at level h.

To address the first case, the LH-AHMCS protocol makes use of both the tail-pointer as well as the `contentionCounter` when inferring whether there is contention within a subdomain. A thread from a subdomain D that enqueues at level $h+1$ infers contention if either the designated `contentionCounter` at level h does not match between acquisition to release or the tail pointer at level h is non-null (on entry on exist). This technique allows a thread targeting level $h+1$ to recognize the presence of peers at an inner domain at level h.

One need not address the second case since a thread, `T`, that falsely assumes no contention at level h, may proceed to level $h+1$ but eventually notices its peers at level h either by virtue of the `contentionCounter` (if a thread was enqueuing at level $h+1$) or by observing a non-null the tail pointer at level h (if a thread had receded to level h in the mean time). In either case, thread `T` would eventually recede to level h. To avoid ping-ponging between two levels when only two threads are present and straddled across two levels, we impose different thresholds for moving up in the tree versus moving down in the tree. The threshold used to decide when to move to a higher level (`MOVE_UP`) is set usually twice the value of the threshold used to move down to a lower level (`MOVE_DOWN`). Different thresholds ensure that two threads do not keep circling between two consecutive levels. Figure 4.25 shows the use of a counter to drive threads of the same domain to a lower level.

Threads straddle multiple non-adjacent levels: The situation where one thread (say $\mathtt{T}_1$) enqueues at level h and another thread (say $\mathtt{T}_2$) from the same subdomain enqueues at level $h+x$, where $x > 1$, is trivially handled. In this case, $\mathtt{T}_1$'s acquisition will progress through the HMCS tree and eventually acquire the lock at level $h+x-1$. Consequently, $\mathtt{T}_2$ will notice the tail pointer being non-null at level $h+x-1$, which makes $\mathtt{T}_2$ progressively recede to lower levels in the tree. In the mean time, $\mathtt{T}_1$, might progress to higher levels in the tree ranging from $[h+1,\ h+x-1]$. Once the two threads settle at a level, both will eventually recede to the lowermost descendent level in the tree where they can take advantage of locality.



(a) Initial state (b) After acquire (c) Round 2...N (d) Round N+1

Figure 4.25 : Figure 4.25(a) represents the initial state, where threads from different domains are intermixed at level h. A thread (say T_1) brings a node X and wants to begin its acquire protocol at level h. A node "A"enqueued at level h belongs to the same domain as T_1. The LH-AHMCS lock enqueues T_1's node X at level h, as shown in Figure 4.25(b). T_1 recognizes the contention arising from the same subdomain at level $h-1$ via the shared counter. In round 2 through N, hysteresis still suggests h as the level for T_1 to begin its acquisition process, as shown in Figure 4.25(c). T_1 enqueues X at level h until round N, each time noticing contention from its subdomain at level $h-1$. In round N+1, hysteresis drives T_1 to target level $h-1$, as shown in Figure 4.25(d).

Property	EagerHysteresis-AHMCS	LazyHysteresis-AHMCS
Length of hysteresis	Only the previous acquisition	Previous N acquisitions
Reactivity to changing contention	Fast	Slow
Resistance to sporadic changes	Low	High

Table 4.8 : Comparison of the eager vs. lazy adaptive HMCS locks

Comparison of eager vs. lazy strategies: We compare various properties of the eager and lazy algorithms in Table 4.8. The eager approach is more reactive to the changing contention and hence best suited for the workloads where the contention changes rapidly. The lazy approach is slow to respond to the changing contention but likely to offer higher throughput for a given contention level if the rate of change of contention is low.

4.9.3 Overlaying fast-path atop hysteresis in AHMCS

The fast-path mechanism and hysteresis are complimentary. The hysteresis can bypass lower levels of an N-level lock hierarchy. The fast-path mechanism can bypass upper levels of an N-level lock hierarchy. In combination, they can bypass any N-1 levels in an N-level hierarchy.

We overlay the fast-path mechanism atop the hysteresis-based adaptation to yield our **F**ast-**P**ath augmented **A**daptive HMCS (FP-AHMCS) lock. We use the general term FP-

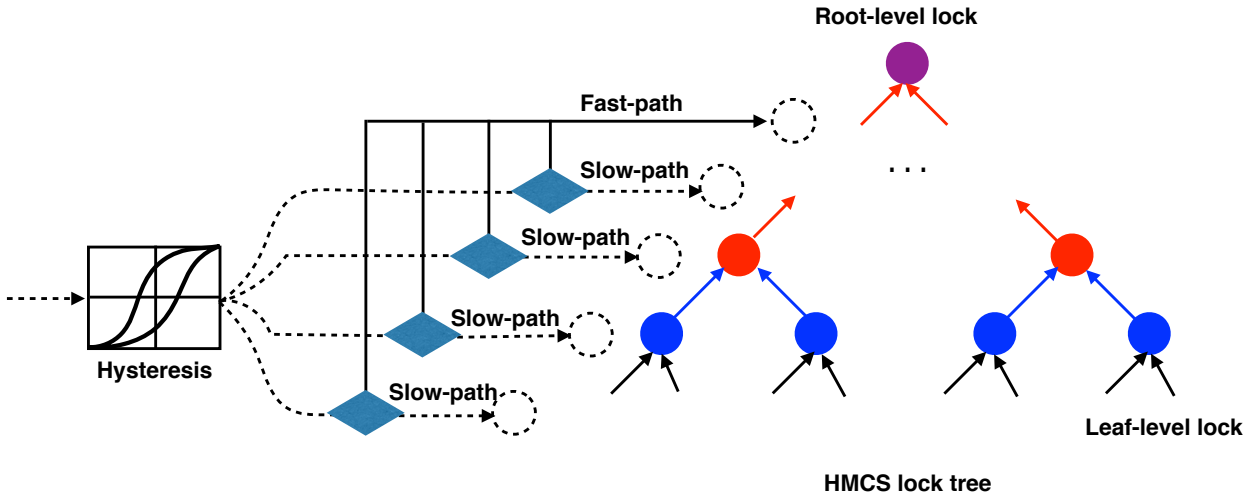


Figure 4.26 : Fast-path augmented Adaptive HMCS lock. A hysteresis mechanism, based on the previously observed contention, suggests the level h where the enqueuing process will begin. If the root-level lock is uncontended, the protocol takes the fast-path and directly enqueues at the root level; otherwise it follows the slow-path through the HMCS$\langle h \rangle$ protocol. Based on the current contention, hysteresis decides whether the next round of acquisition should suggest level h, $h+1$, or $h-1$.

AHMCS when the reference does not distinguish eager vs. lazy implementation choices of AHMCS locks. We call the fast-path augmented EH-AHMCS lock as FP-EH-AHMCS lock. We call the fast-path augmented LH-AHMCS lock as FP-LH-AHMCS lock. In this enhancement, hysteresis guides the level where a thread initially arrives. If that level is uncontended (the tail pointer of the MCS lock at that level is null), we check if the root-level is uncontended. If both the current level and the root level are uncontended, then we directly enqueue at the root level (fast-path), often immediately acquiring the lock. Subsequently, we release from the root level. If the root level is contended, we follow either the eager or the lazy strategy to enqueue at an appropriate level and use the feedback of the acquisition to drive the next round of acquisition. Figure 4.26 shows the schematic diagram of an FP-AHMCS lock. Each diamond shape represents the entire logic of the two diamonds of the fast-path mechanism shown in Figure 4.19.

4.10 Hardware transactional memory with AHMCS

Emerging architectures such as IBM POWER8 [127], IBM Blue Gene/Q [234], and Intel's Haswell [195] offer hardware support for transactions, often referred to as Hardware Transactional Memory (HTM). In this section, we explore using HTM to achieve the fast-path in an AHMCS lock.

HTM offers a set of primitives to achieve an optimistic concurrency. Each thread *starts* a transaction assuming it will be able to update a shared data without concurrent accesses by any other thread. When a thread updates a shared datum inside a transaction, all updates are kept local so that other threads do not notice its changes. If a thread completes all operations in its transaction without a conflicting access by any other thread, it *commits* the transaction, and all updates become visible to the other threads atomically. However, if an access conflicts with a concurrent access by another thread, the hardware *aborts* the transaction discarding all changes made during the transaction. A programmer can manually force a transaction to abort if desired. Any HTM support offers at least the following three primitives:

1. Begin transaction: starts a region of a transaction. This primitive offers the functionality analogous to the `setjmp` routine in C.

2. Commit transaction: ends and attempts to commit a transaction previously started.

3. Abort transaction: discards all local changes. This primitive offers the functionality analogous to the `longjmp` routine in C. When a transaction fails or aborts, the processor restores its architectural state that existed just before the beginning of the transaction. Similar to the `longjmp`, a conditional code indicates the failure to complete a transaction, which can be used to change the control flow.

Due to its optimistic concurrency feature, the HTM is an attractive alternative to replace locks. Since transactions abort due to conflicts, naively retrying a transaction can cause starvation. Furthermore, transactions do not ensure the locality of reference. One cannot, however, do away with locks; HTM approaches use global locks as a fallback mechanism when a transaction fails.

Algorithm 5: HTM-AHMCS protocol

```
1  ThreadLocal boolean tlTookFastPath /* Initially false                          */
2  void Acquire(...)
3      if IsLeafLevelLockFree() then
4          if BEGIN_TRANSACTION() == SUCCESS then
5              if IsRootLevelLockFree() then
6                  tlTookFastPath ⟵ true
7                  return                                          /* Fast-path */
8              else
9                  ABORT_TRANSACTION()
10         /* Abort and transaction failures land here                          */
11     Start AHMCS slow-path Acquire                          /* Fallback slow-path */
12 void Release(...)
13     if tlTookFastPath then
14         COMMIT_TRANSACTION()
15         tlTookFastPath ⟵ false
16         return
17     Start AHMCS slow-path Release
```

We substitute our fast-path mechanism with an HTM analog to make use of optimistic concurrency. In this design, a thread—wanting to acquire the lock—begins a transaction. It then checks if the root-level lock is free (tail pointer is null); if so, it enters the critical section, executing it as a transaction. If the root-level lock is not free or the transaction performs conflicting memory accesses, then the transaction aborts, and the protocol falls back to the slow-path route of the AHMCS lock (either eager or lazy protocols).

4.10.1 AHMCS algorithm with HTM

We present the AHMCS lock with the HTM in Algorithm 5. We refer to this lock as HTM-AHMCS since it does not use the software solution for fast-path. If the underlying AHMCS lock is EH-AHMCS, we call such lock as HTM-EH-AHMCS lock. If the underlying AHMCS lock is LH-AHMCS, we call such lock as HTM-EH-AHMCS lock. Similar to the fast-path mechanism, we set a thread-local flag `tlTookFastPath` if the fast-path is taken and unset it if the transaction succeeds; if the transaction fails, the hardware reverts the value of `tlTookFastPath`.

4.10.2 Correctness

If two threads are simultaneously inside a transaction, then the hardware aborts at least one of them if they have a conflicting access; otherwise, both may succeed if they do not perform conflicting accesses. If two threads take the slow-path through the AHMCS lock, then the mutual exclusion between them is naturally enforced in software. If one thread (T_{htm}) takes the fast-path route via hardware transactions and another thread (T_{ahmcs}) takes the slow-path via the AHMCS lock, the following scenarios happen:

1. T_{htm} checks the root-level tail pointer to be null, finishes its critical section, and successfully commits its transactions before T_{ahmcs} proceeds to acquire the root-level lock (updates the tail pointer). This interleaving ensures mutual exclusion between T_{htm} and T_{ahmcs}.

2. T_{ahmcs} updates the root-level tail pointer before T_{htm} checks it. In this case, T_{htm} aborts the fast-path voluntarily. This interleaving ensures that only T_{ahmcs} performs its critical section.

3. T_{ahmcs} updates the root-level tail pointer after T_{htm} checks it. In this case, the hardware recognizes the conflicting access to the same root-level tail pointer and aborts T_{htm} immediately. This interleaving ensures that only T_{ahmcs} performs its critical section.

4.10.3 Performance

As mentioned in Section 4.9.1, checking the root-level tail pointer does not create remote traffic when the lock is contended since 1) the root-level tail pointer is typically cached, and 2) the root-level tail pointer changes less frequently. In addition, to avoid checking the root-level tail pointer frequently, we first check the current-level tail pointer; if it is non-null, we take the slow-path directly without starting a transaction. Finally, when the lock is not contended, checking the root-level tail pointer allows the protocol to typically succeed in its transaction, and hence proves beneficial.

4.11 Evaluation of adaptive locks

In this section, we evaluate several variants of adaptive HMCS locks presented in the previous sections and compare them with HMCS locks. For most of our experiments, we use an IBM Power 755. We use SGI UV 1000 on a few occasions. For a limited study of HTM, we use an IBM POWER8 machine.[8] This section is not concerned with assessing the correct value of passing threshold; earlier sections have already evaluated the passing threshold. Hence, all experiments in this section set the passing thresholds at all levels (if needed) to a fixed value—64. In the context of this section, an "ideal' or a "best" lock refers to the best performing lock within a set of possible HMCS lock variants; it should not be confused with a "universally" best lock.

Unlike the evaluation in Section 4.8, for all experiments in this section, we bind the threads *sparsely*. With the sparse thread binding, when there are 4 threads in the experiments, each thread binds to an independent socket of a Power 755; when there are 8 threads in the experiments, 2 threads share a socket but each thread binds to a different core; when there are 64 threads in the experiments, 16 threads share a socket, 2 threads each share a core, but each thread binds to a different SMT thread. On a Power 755, the ideal lock for the contention levels of 1, 2, and 4 threads is HMCS$\langle 1 \rangle$; the ideal lock for the contention levels of 8, 16, and 32 threads is HMCS$\langle 2 \rangle$; and the ideal lock for the contention levels of 64 and 128 threads is HMCS$\langle 3 \rangle$. In the context of this section, a shallower HMCS lock ignores the nearer neighbors; that is an HMCS$\langle 2 \rangle$ is a lock formed by ignoring the possibility of passing within the SMT threads.

On SGI UV 1000, we use only 256 threads (8 nodes) and form a 4-level hierarchy. With the sparse thread binding, when there are 4 threads in the experiments, each thread binds to an independent pair of nodes connected via NUMAlink [208] technology; when there are 8 threads in the experiments, each thread binds to a different blade with two of them per QPI-connected [97] blade pair; when there are 32 threads in the experiment, each thread binds to a different core with two threads per socket; when there are 256 threads, each thread binds to a different hardware thread with two threads per core. The ideal lock for the contention

[8] We defer the details of the IBM POWER8 to Section 4.11.5.

levels of 1, 2, and 4 threads is HMCS⟨1⟩. The ideal lock for the contention levels of 8 and 16 threads is HMCS⟨2⟩. The ideal lock for the contention levels of 32, 64, and 128 threads is HMCS⟨3⟩. The ideal lock for the contention level of 256 threads is HMCS⟨4⟩.

In Section 4.11.1 we evaluate the fast-path mechanism (FP-HMCS). In Section 4.11.2 we evaluate the hysteresis used in AHMCS locks (EH-AHMCS and LH-AHMCS). In Section 4.11.3 we evaluate the combined effect of the fast-path and hysteresis (FP-EH-AHMCS and FP-LH-AHMCS). In Section 4.11.4 we evaluate the sensitivity of AHMCS for fast changing contention levels. In Section 4.11.5 we evaluate the HTM combined with the AHMCS lock (HTM-AHMCS).

4.11.1 Utility of a fast-path

The following two aspects of the fast-path mechanism are of importance:

1. The throughput difference between a fast-path augmented HMCS⟨3⟩ (abbreviated as FP-HMCS⟨3⟩) and an HMCS⟨1⟩ under *no contention*, and

2. The throughput difference between a FP-HMCS⟨3⟩ and an HMCS⟨3⟩ under *full contention*.

We set up a controlled experiment, where we assess the above two factors. In our experiments, we vary the critical section size from 0-64 cache lines updates. Each thread has no delay outside the critical section. Note that in this setup hysteresis is not used.

Table 4.9 compares the throughput of FP-HMCS⟨3⟩ and HMCS⟨1⟩ at 1 thread (no contention). Table 4.10 compares the throughput of FP-HMCS⟨3⟩ and HMCS⟨3⟩ at 128 threads (full contention). In this setup, the lock throughput under no contention includes the round trip time and hence captures the latency of lock acquisition. The lock throughput under full contention hides the delay outside the critical path and hence captures the throughput of lock acquisition. Clearly, the fast-path mechanism helps achieve close to peak performance (92% or more) under no contention and maintains high throughput under full contention (92% or more).

A few configurations of FP-HMCS⟨3⟩ deliver more than 1× of the peak performance. This anomaly could be due to measurement errors, hardware prefetching effects, and turbo

Cache lines written in the critical section	Throughput (locks/sec) HMCS⟨1⟩	FP-HMCS⟨3⟩	Fraction of peak throughput FP-HMCS⟨1⟩ / HMCS⟨1⟩
0	2.18E+07	2.00E+07	0.92×
1	2.22E+07	2.08E+07	0.94×
2	2.16E+07	2.03E+07	0.94×
4	1.69E+07	1.57E+07	0.93×
8	1.02E+07	1.05E+07	1.03×
16	3.15E+06	3.75E+06	1.19×
32	8.02E+05	2.05E+06	2.56×
64	4.37E+05	6.98E+05	1.60×

Table 4.9 : A fast-path augmented HMCS⟨3⟩ delivers more than 92% of HMCS⟨1⟩'s peak throughput under no contention.

Cache lines written in the critical section	Throughput (locks/sec) HMCS⟨3⟩	FP-HMCS⟨3⟩	Fraction of peak throughput FP-HMCS⟨3⟩ / HMCS⟨3⟩
0	2.65E+07	2.48E+07	0.93×
1	2.51E+07	2.43E+07	0.97×
2	2.40E+07	2.19E+07	0.92×
4	1.49E+07	1.66E+07	1.11×
8	9.24E+06	9.06E+06	0.98×
16	3.69E+06	3.40E+06	0.92×
32	9.02E+05	1.08E+06	1.19×
64	2.46E+05	2.55E+05	1.04×

Table 4.10 : A fast-path augmented HMCS⟨3⟩ maintains more than 92% of HMCS⟨3⟩'s peak throughput under full contention.

boost, among others. We do not investigate the root causes of such anomalous performance benefits.

Figure 4.27 plots the throughput graphs of HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, and FP-HMCS⟨3⟩ locks at various contention levels over a range of critical section sizes. Other than the no contention cases (1 thread), the fixed-depth HMCS⟨3⟩ lock delivers close to ideal throughput on several occasions, especially once the size of the critical section increases. This behavior is because the overhead of additional lock acquisition in HMCS⟨3⟩ does not appear on the critical path either when the contention is heavy or when the critical section is large. Since the FP-HMCS⟨3⟩ lock overcomes the overhead of HMCS⟨3⟩ under no contention, it is a valuable optimization to add to the HMCS lock. At larger critical section sizes, 128 threads (four SMTs per core) seem to do worse than 64 threads (two SMTs per core) for all locks. This anomaly is likely because a core's L1 cache bandwidth saturates with three hardware threads polling three different memory locations plus the lock-holding SMT thread streaming multiple cache lines.

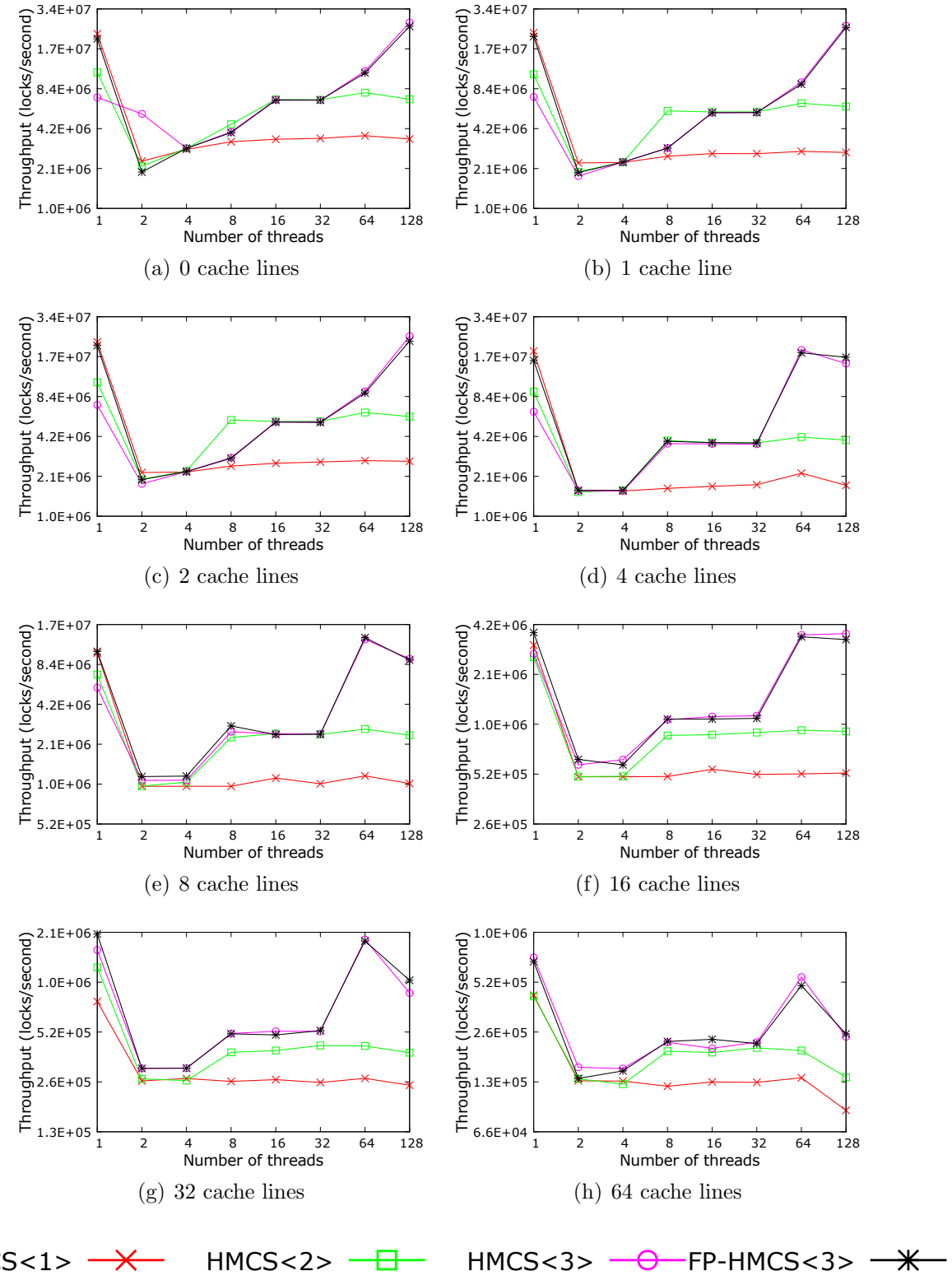


Figure 4.27 : Comparison of HMCS⟨1⟩, HMCS⟨2⟩, and HMCS⟨3⟩ with FP-HMCS⟨3⟩ on Power 755.

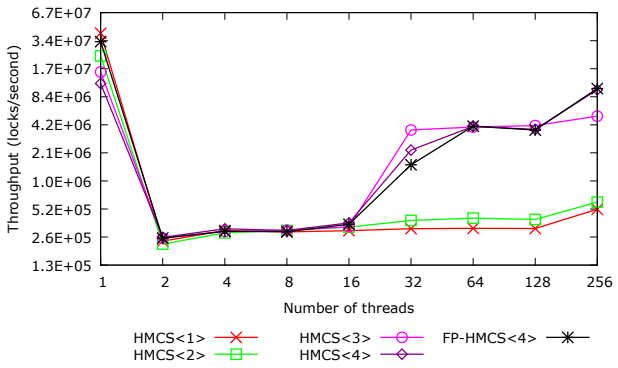


Figure 4.28 : Throughput comparison of HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, and HMCS⟨4⟩ locks with FP-HMCS⟨4⟩ on SGI UV 1000.

To understand the limitations of the fast-path mechanism, we highlight a particular case where the fast-path mechanism fails to match the throughput of an ideal HMCS lock at moderate contentions. At moderate contention, with eight threads and small critical section sizes such as two or four cache line updates (Figure 4.27(b) and (c)), FP-HMCS⟨3⟩ is unable to match the throughput of HMCS⟨2⟩. In these configurations, HMCS⟨2⟩ is the right choice since it exploits the local passing within a socket but does not introduce the overhead of an additional core-level lock used in an HMCS⟨3⟩.

As yet another data point, we evaluated an FP-HMCS⟨4⟩ lock on SGI UV 1000 with 256 threads (8 nodes) and compared it with HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, and HMCS⟨4⟩ locks. In this setup, the critical section performs four cache line updates. The FP-HMCS⟨4⟩ fails to match the throughput of an HMCS⟨3⟩ at 32 threads (Figure 4.28). This difference is because the fast-path mechanism (without the hysteresis) can only expedite the path from a leaf to the root of the tree. Clearly, the fast-path mechanism without the hysteresis is handicapped in the cases where an intermediate level of the tree is appropriate for starting the acquisition process. The same behavior is seen for two cache line updates as well (not shown).

4.11.2 Benefits of hysteresis

In this section, we evaluate the hysteresis mechanism used in EH-AHMCS and LH-AHMCS locks. Note that the fast-path mechanism is not present in these locks. We employ two

No. threads	Percentage of acquisitions at			Ideal level
	level 1 (leaf)	level 2 (middle)	level 3 (root)	of acquisition
1	0.00%	0.00%	**100.00%**	3 (root)
2	0.00%	0.00%	**100.00%**	3 (root)
4	0.00%	0.00%	**100.00%**	3 (root)
8	0.00%	**99.96%**	0.04%	2 (middle)
16	0.00%	**99.99%**	0.01%	2 (middle)
32	0.00%	**100.00%**	0.00%	2 (middle)
64	**100.00%**	0.00%	0.00%	1 (leaf)
128	**100.00%**	0.00%	0.00%	1 (leaf)

Table 4.11 : Percentage of lock acquisitions at different levels for the EH-AHMCS lock.

No. threads	Percentage of acquisitions at			Ideal level
	level 1 (leaf)	level 2 (middle)	level 3 (root)	of acquisition
1	0.00%	0.00%	**100.00%**	3 (root)
2	0.00%	0.00%	**100.00%**	3 (root)
4	0.00%	0.00%	**100.00%**	3 (root)
8	0.00%	**99.97%**	0.03%	2 (middle)
16	0.00%	**100.00%**	0.00%	2 (middle)
32	0.00%	**100.00%**	0.00%	2 (middle)
64	**100.00%**	0.00%	0.00%	1 (leaf)
128	**100.00%**	0.00%	0.00%	1 (leaf)

Table 4.12 : Percentage of lock acquisitions at different levels for the LH-AHMCS lock.

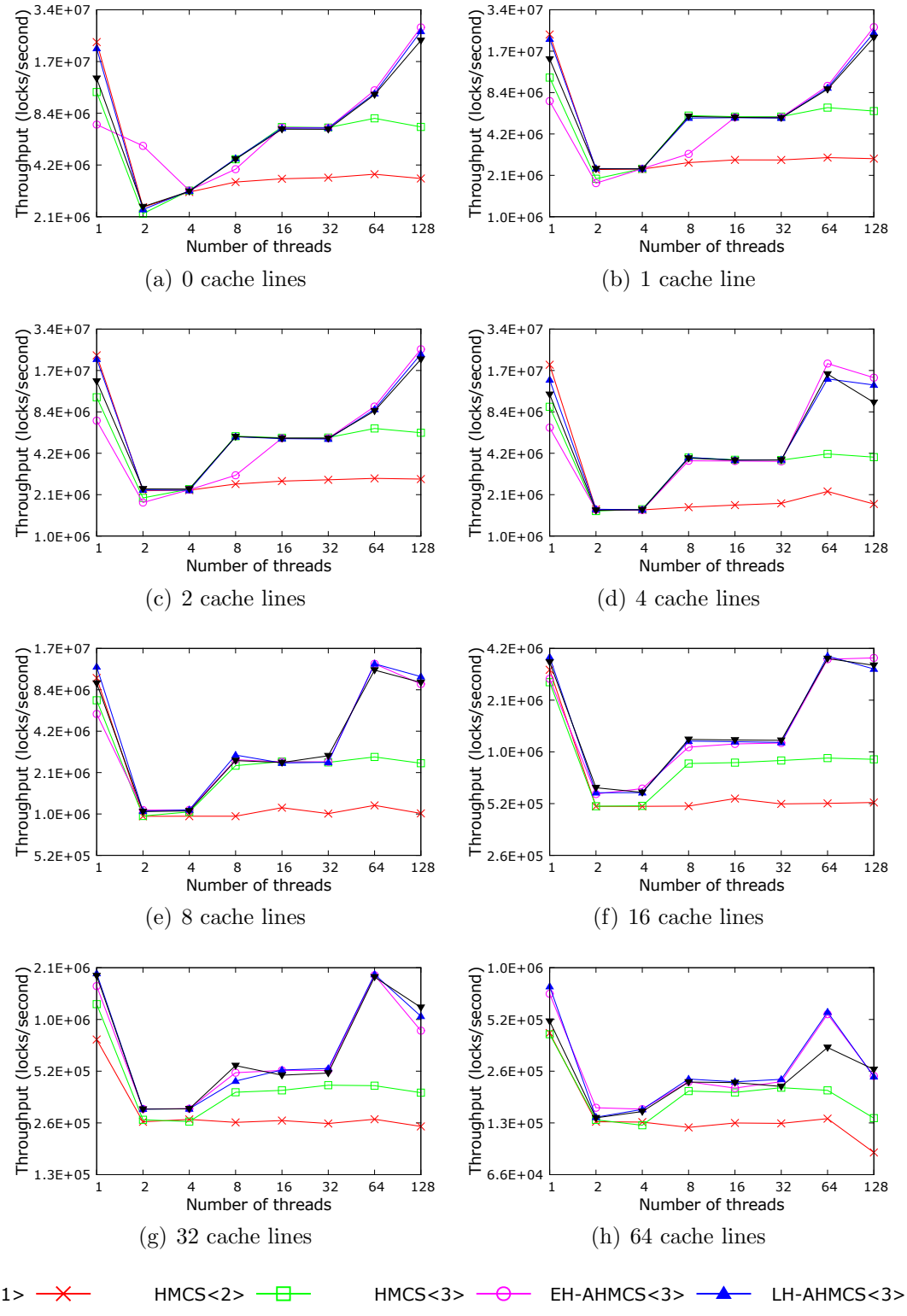
techniques for our evaluation, one for the precision of hysteresis to choose a level and another for the performance impact of hysteresis.

First, to better understand the precision of hysteresis, we profile where the AHMCS lock acquisition process begins at different contention levels in the 3-level locking hierarchy on Power 755. We vary the contention by changing the number of participating threads. We use eight different contention levels depending on the number of threads, i.e., 1, 2, 4, 8, 16, 32, 64, and 128. Once a contention level is chosen for an execution, we do not change it again within the same run. We give 30 seconds for each experimental run, which is a reasonably long period for an adaptive lock to identify the contention and deliver its best possible performance. In our experiments, we fix the critical section to two cache line updates. Each thread has no delay outside the critical section. Table 4.11 and Table 4.12 show the distribution of where the acquisition process begins in the hierarchy at different contention levels for EH-AHMCS and LH-AHMCS locks, respectively. We infer that both eager and lazy hysteresis approaches skip the unnecessary levels of the hierarchy under ideal circumstances.

Second, to assess the performance impact of hysteresis, we measure the throughput of EH-AHMCS and LH-AHMCS locks. We vary the critical section from 0-64 cache line updates. As before, we give 30 seconds for each experimental run, which is a reasonably long period for an adaptive lock to identify the contention and deliver its best possible performance.

Figure 4.29 compares the performance of HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, EH-AHMCS⟨3⟩, and LH-AHMCS⟨3⟩ for different levels of contention and different lengths of the critical sections. We observe that the throughput of both EH-AHMCS⟨3⟩ and LH-AHMCS⟨3⟩ locks match that of the best HMCS lock at various levels of contention. An only exception is for the case of 0 cache lines with two threads, where HMCS⟨3⟩ has a superior performance. This behavior is, in fact, anomalous since HMCS⟨3⟩ behaves better than HMCS⟨1⟩ and HMCS⟨2⟩ at low contention. This anomaly happens because an HMCS⟨3⟩ does not encounter as much contention at this level as faced the by other locks. In this case, HMCS⟨3⟩'s critical path is shorter and its delay outside the critical path is relatively larger. A thread that acquires the root-level lock often finds no successor or a predecessor because the other thread has a longer path, which involves: 1) releasing its lower two levels of locks, 2) reinitializing its `QNodes`, and 3) reacquiring the lower two levels, before arriving to acquire the root level lock. Where there is no predecessor, the MCS lock acquire protocol does not access its `status` and its predecessor's `next` fields; as a result, the HMCS⟨3⟩ lock avoids the overhead of accessing a few remote cache lines. The EH-AHMCS⟨3⟩ and LH-AHMCS⟨3⟩ locks behave as expected by eliminating the leaf-level lock and starting the acquisition protocol at level two. The adaptive lock is unable to match this one-off anomalously superior performance of the HMCS⟨3⟩.

This experiment confirms that given sufficient time, EH-AHMCS and LH-AHMCS locks recognize the contention and adjusts appropriately, usually delivering the performance of the best lock in a particular configuration. Furthermore, it shows that hysteresis can "right-size" a lock for any contention level. We summarize the quantitative performance numbers in Table 4.13. On average, EH-AHMCS⟨3⟩ reaches 99% of the throughput of the "ideal" lock over a range of critical section sizes and contention levels. On average, LH-AHMCS⟨3⟩ reaches 94% of the throughput of the "ideal" lock over a range of critical section sizes and contention levels. Compared to an ideal lock, the worst-case throughputs of both eager and



(a) 0 cache lines

(b) 1 cache line

(c) 2 cache lines

(d) 4 cache lines

(e) 8 cache lines

(f) 16 cache lines

(g) 32 cache lines

(h) 64 cache lines

Figure 4.29 : Comparison of HMCS⟨1⟩, HMCS⟨2⟩, and HMCS⟨3⟩ locks with EH-AHMCS⟨3⟩ and LH-AHMCS⟨3⟩ locks on Power 755.

Lock		Fraction of peak performance compared to the best lock	
		Average (geometric mean) (higher is superior)	Worst case (higher is superior)
Fixed depth HMCS	HMCS⟨1⟩	0.47×	0.09×
	HMCS⟨2⟩	0.68×	0.21×
	HMCS⟨3⟩	0.89×	0.33×
Fast-path only	FP-HMCS⟨3⟩	0.97×	0.51×
Hysteresis only	EH-AHMCS⟨3⟩	0.99×	0.77×
	LH-AHMCS⟨3⟩	0.94×	0.61×
Fast-path and Hysteresis	FP-EH-AHMCS⟨3⟩	1.00×	0.78×
	FP-LH-AHMCS⟨3⟩	0.99×	0.77×

Table 4.13 : Performance comparison of HMCS, Fastpath-HMCS, Hysteresis-AHMCS and Fastpath-Hysteresis-AHMCS locks. The summary is computed over the critical section length ranging from 0 to 64 cache line updates and contention varying from 0 to 128 threads.

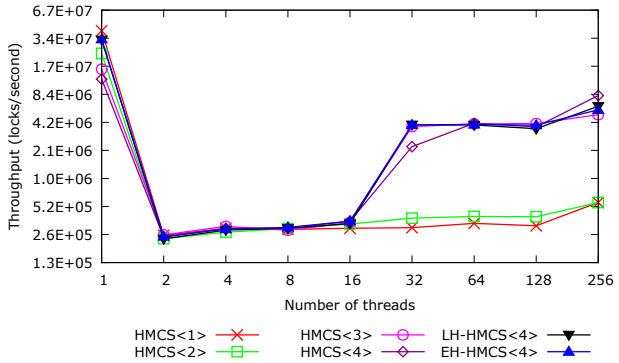


Figure 4.30 : Comparison of HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, and HMCS⟨4⟩ locks with EH-AHMCS⟨4⟩ and LH-AHMCS⟨4⟩ on SGI UV 1000.

lazy hysteresis-based locks in any configuration (excluding the anomalous case of 2 threads and zero critical section length) are much superior to the worst-case throughput of any fixed-depth HMCS lock.

For completeness, we show that the hysteresis was able to address the previously mentioned performance gap on SGI UV 1000 also. Both EH-AHMCS⟨4⟩ and LH-AHMCS⟨4⟩ locks match the performance of HMCS⟨3⟩ at 32 threads (Figure 4.30).

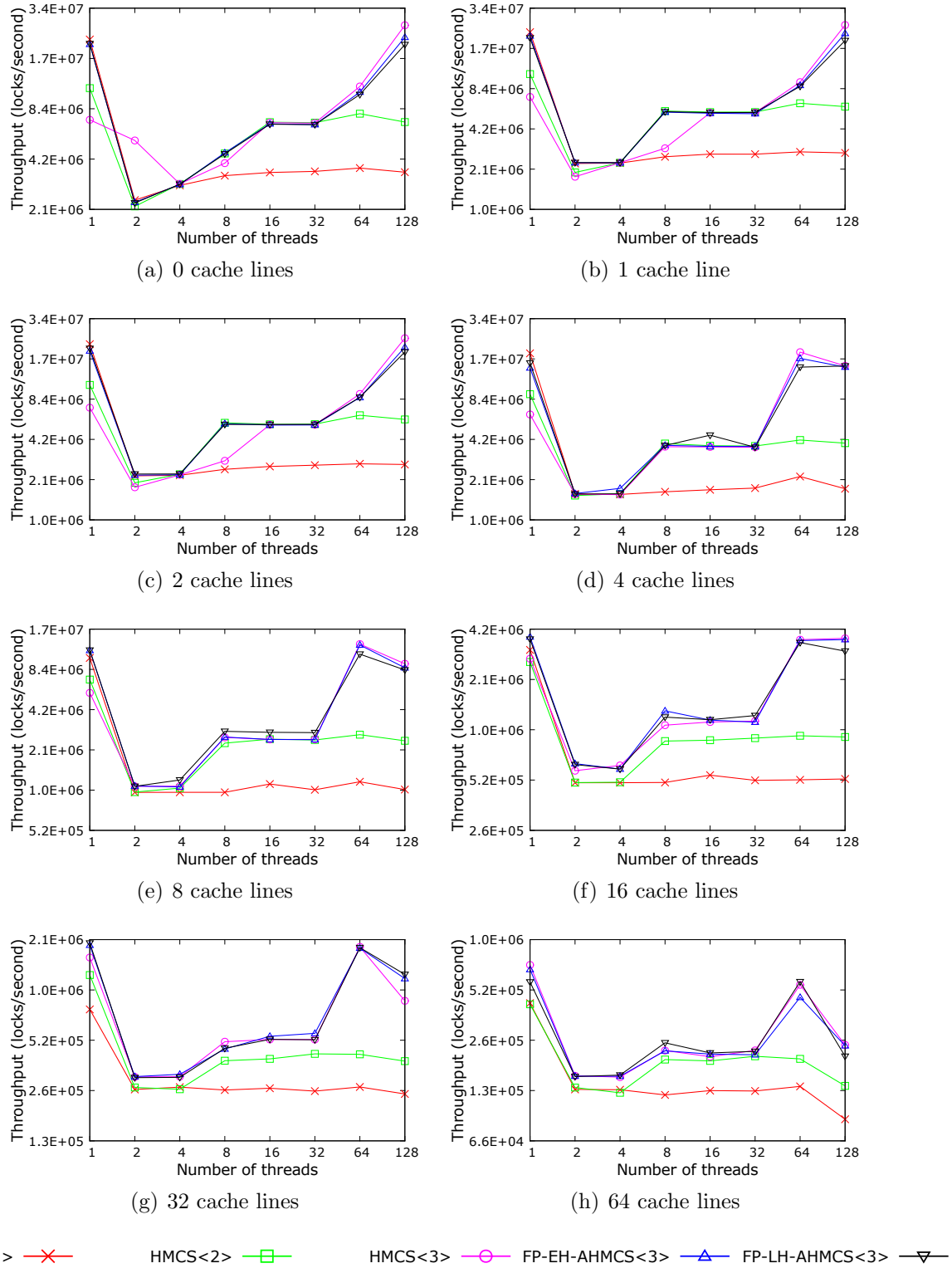
4.11.3 Value of hysteresis and a fast-path

An obvious limitation of the hysteresis is its inability to adjust quickly from high contention to low contention and vice-versa. An obvious limitation of the fast-path is its inability to handle certain contention levels. The AHMC lock, which combines both fast-path and

hysteresis, ameliorates these limitations. We assess both eager and lazy hysteresis approaches under ideal circumstances in this subsection.

Figure 4.31 compares the performance of HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, FP-EH-AHMCS⟨3⟩, and FP-LH-AHMCS⟨3⟩ for different levels of contention and different critical section sizes. As before, we give 30 seconds for each experimental run, which is a reasonably long period for an adaptive lock to identify the contention and deliver its best possible performance. FP-EH-AHMCS⟨3⟩ and FP-LH-AHMCS⟨3⟩ locks match the throughput of the best HMCS lock at various levels of contention. An exception is the case of 0 cache lines with two threads for the previously described reason. The performance of FP-AHMCS locks closely follows the performance of their AHMCS counterparts without a fast-path. *This experiment confirms that overlaying the fast-path at each level in an AHMCS lock does not add a significant overhead.*

We summarize the quantitative performance numbers in Table 4.13. On average, FP-EH-AHMCS⟨3⟩ reaches the performance of an "ideal" lock over a range of critical section sizes and contention levels; EH-AHMCS⟨3⟩ reaches 99% of the peak. On average, FP-LH-AHMCS⟨3⟩ reaches 99% of the performance of an "ideal" lock over a range of critical section sizes and contention levels; LH-AHMCS⟨3⟩ reaches 94% of the peak. Compared to an ideal lock, the worst-case throughputs of both FP-AHMCS locks in any configuration (excluding the anomalous case of 2 threads and zero critical section length) are much superior to the worst-case throughput of fixed-depth HMCS locks. Moreover, FP-AHMCS locks closely follow their AHMCS⟨3⟩ counterparts without a fast-path.



(a) 0 cache lines (b) 1 cache line

(c) 2 cache lines (d) 4 cache lines

(e) 8 cache lines (f) 16 cache lines

(g) 32 cache lines (h) 64 cache lines

Figure 4.31 : Comparison of HMCS⟨1⟩, HMCS⟨2⟩, and HMCS⟨3⟩ locks with FP-EH-AHMCS⟨3⟩ and FP-LH-AHMCS⟨3⟩ locks on Power 755.

4.11.4 Sensitivity to variable contention

In this subsection, we evaluate the sensitivity of several (HMCS, FP-HMCS, and FP-AHMCS) locks. We use eight different contention levels depending on the number of threads (i.e., 1, 2, 4, 8, 16, 32, 64, and 128). We randomly exercise all possible transitions from one contention level to another. To assess the sensitivity of our adaptive locks, we change the contention from one level to another at six different frequencies: 1 microsecond, 10 microseconds, 100 microseconds, 1 millisecond, 10 milliseconds, and 100 milliseconds. In this test setup, unlike the previous one, we vary the number of threads within the same execution. We do not vary the frequency with which the contention changes within the same execution. We study the sensitivity with two different sizes of critical sections: two cache line updates and four cache line updates. We define the following terminology with respect to this experiment:

Definition 4.4 (Fractional throughput) *Fractional throughput $\mathcal{F}_k(c)$ of a lock k at a contention level c is the ratio of its observed throughput $\mathcal{T}_k(c)$ at contention level c and the maximum throughput of any fixed-depth HMCS lock at the same contention level.*

$$\mathcal{F}_k(c) = \frac{\mathcal{T}_k(c)}{\max\limits_{1 \leq j \leq n} \Big(\mathcal{T}_{hmcs\langle j \rangle}(c) \Big)} \tag{4.40}$$

Definition 4.5 (Mean fractional throughput) *Mean fractional throughput $\mathcal{M}_k$ of a lock k is the geometric mean of its fractional throughput $\mathcal{F}_k(c)$ observed over various contention levels. If an experiment exercises a sequence of N (not necessarily unique) contention levels, then,*

$$\mathcal{M}_k = \Big(\prod_{i=1}^{N} \mathcal{F}_k(c_i) \Big)^{1/N} \tag{4.41}$$

Definition 4.6 (Worst-case fractional throughput) *Worst-case fractional throughput $\mathcal{W}_k$ of a lock k is the minimum of its fractional throughput $\mathcal{F}_k(c)$ observed over various contention levels. If an experiment exercises a sequence of N (not necessarily unique) contention levels, then,*

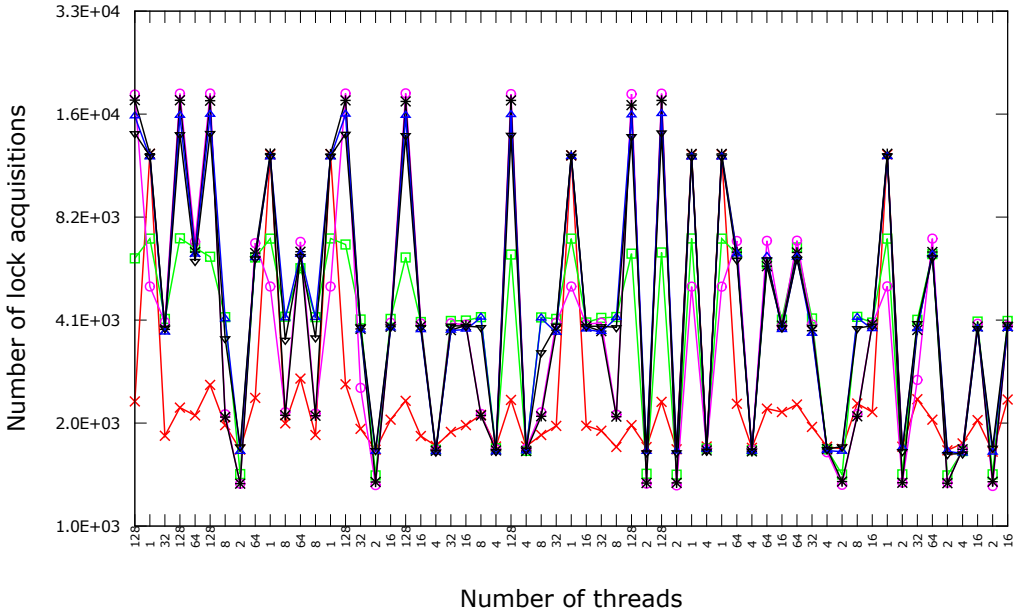
$$\mathcal{W}_k = \min_{1 \leq i \leq N} \mathcal{F}_k(c_i) \tag{4.42}$$

Informally, $\mathcal{M}$ quantifies the overall behavior of a lock whereas $\mathcal{W}$ quantifies the worst-case behavior of a lock over a range of contention levels. Higher values of $\mathcal{M}$ and $\mathcal{W}$ imply a superior lock.

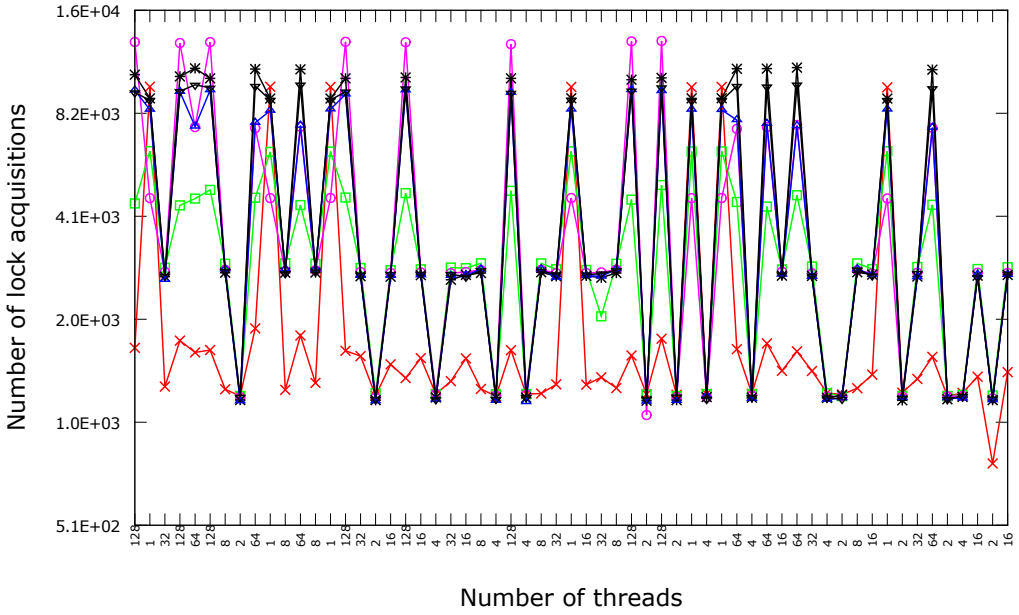
Figure 4.32(a) shows the throughput of various locks (HMCS⟨1⟩, HMCS⟨2⟩, HMCS⟨3⟩, FP-HMCS⟨3⟩, FP-EH-AHMCS⟨3⟩, and FP-LH-AHMCS⟨3⟩) with two cache lines being updated in the critical section, when we change the contention every 100 microseconds. Figure 4.32(b) shows the same with four cache lines being updated in the critical section.

Summary for two cache line updates: Table 4.14 shows the *mean fractional throughput* $\mathcal{M}$ of various locks over a range of contention periods. The same information is graphically presented in Figure 4.33(a). Table 4.15 shows the *worst-case fractional throughput* $\mathcal{W}$ of various locks over a range of contention periods. The same information is graphically presented in Figure 4.33(b).

Summary for four cache line updates: Table 4.16 shows the *mean fractional throughput* $\mathcal{M}$ of various locks over a range of contention periods. The same information is graphically presented in Figure 4.34(a). Table 4.17 shows the *worst-case fractional throughput* $\mathcal{W}$ of various locks over a range of contention periods. The same information is graphically presented in Figure 4.34(b).



(a) Critical section with 2 cache line updates



(b) Critical section with 4 cache line updates

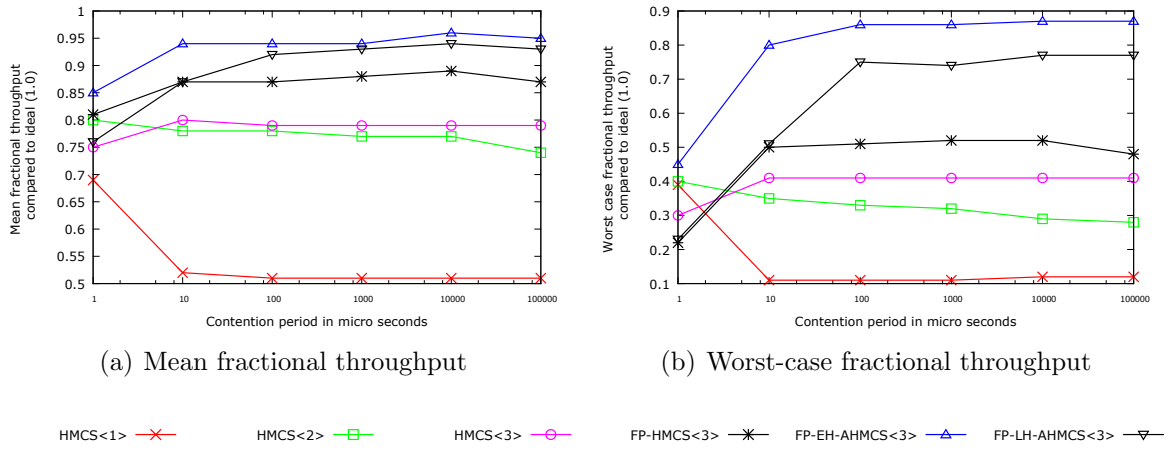
Figure 4.32 : Lock throughput with contention changing every 100μs period. The X axis represents the contention (number of threads) at each 100μs period. The Y axis represents the number of locks acquired in each 100μs interval

Period of change of contention	Mean fractional throughput $\mathcal{M}$ (higher is superior)					
	HMCS⟨1⟩	HMCS⟨2⟩	HMCS⟨3⟩	FP-HMCS⟨3⟩	FP-EH-AHMCS⟨3⟩	FP-LH-AHMCS⟨3⟩
10^0 μ sec	0.69×	0.80×	0.75×	0.81×	0.85×	0.76×
10^1 μ sec	0.52×	0.78×	0.80×	0.87×	0.94×	0.87×
10^2 μ sec	0.51×	0.78×	0.79×	0.87×	0.94×	0.92×
10^3 μ sec	0.51×	0.77×	0.79×	0.88×	0.94×	0.93×
10^4 μ sec	0.51×	0.77×	0.79×	0.89×	0.96×	0.94×
10^5 μ sec	0.51×	0.74×	0.79×	0.87×	0.95×	0.93×

Table 4.14 : Mean fractional throughput of various locks (compared to the best performing lock at the same contention level) over a range of contention levels varying for different periods. Critical section size = 2 cache line updates.

Period of change of contention	Worst-case fractional throughput $\mathcal{W}$ (higher is superior)					
	HMCS⟨1⟩	HMCS⟨2⟩	HMCS⟨3⟩	FP-HMCS⟨3⟩	FP-EH-AHMCS⟨3⟩	FP-LH-AHMCS⟨3⟩
10^0 μ sec	0.39×	0.40×	0.30×	0.22×	0.45×	0.23×
10^1 μ sec	0.11×	0.35×	0.41×	0.50×	0.80×	0.51×
10^2 μ sec	0.11×	0.33×	0.41×	0.51×	0.86×	0.75×
10^3 μ sec	0.11×	0.32×	0.41×	0.52×	0.86×	0.74×
10^4 μ sec	0.12×	0.29×	0.41×	0.52×	0.87×	0.77×
10^5 μ sec	0.12×	0.28×	0.41×	0.48×	0.87×	0.77×

Table 4.15 : Worst-case fractional throughput of various locks (compared to the best performing lock at the same contention level) over a range of contention levels varying for different periods. Critical section size = 2 cache line updates.



(a) Mean fractional throughput

(b) Worst-case fractional throughput

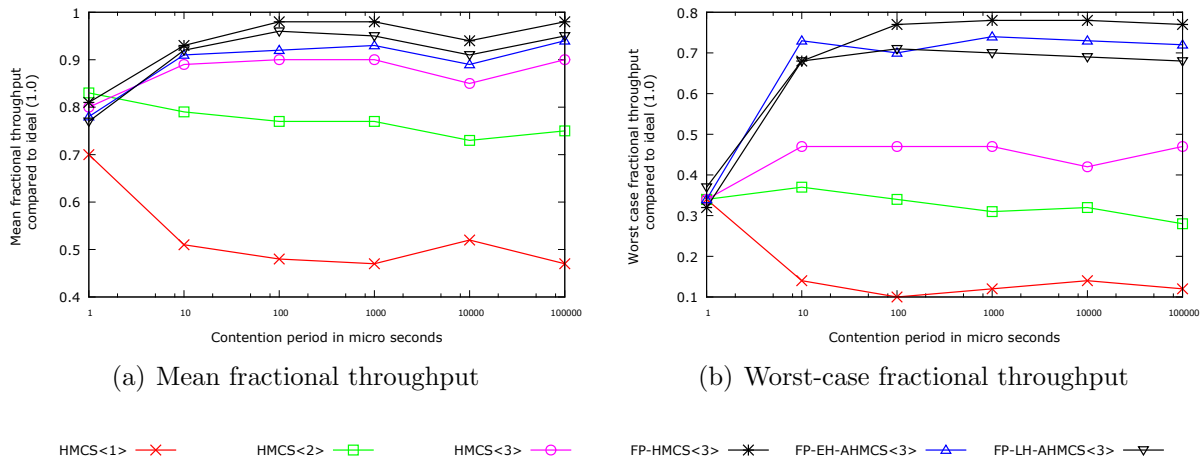
Figure 4.33 : Relative throughput of various locks (compared to the best performing HMCS lock) at different contention levels for different periods of contention. Critical section size = 2 cache line updates.

Period of change of contention	Mean fractional throughput $\mathcal{M}$ (higher is superior)					
	HMCS⟨1⟩	HMCS⟨2⟩	HMCS⟨3⟩	FP-HMCS⟨3⟩	FP-EH-AHMCS⟨3⟩	FP-LH-AHMCS⟨3⟩
10^0 μ sec	0.70×	0.83×	0.80×	0.81×	0.78×	0.77×
10^1 μ sec	0.51×	0.79×	0.89×	0.93×	0.91×	0.92×
10^2 μ sec	0.48×	0.77×	0.90×	0.98×	0.92×	0.96×
10^3 μ sec	0.47×	0.77×	0.90×	0.98×	0.93×	0.95×
10^4 μ sec	0.52×	0.73×	0.85×	0.94×	0.89×	0.91×
10^5 μ sec	0.47×	0.75×	0.90×	0.98×	0.94×	0.95×

Table 4.16 : Mean fractional throughput of various locks (compared to the best performing lock at the same contention level) over a range of contention levels varying for different periods. Critical section size = 4 cache line updates.

Period of change of contention	Worst-case fractional throughput $\mathcal{W}$ (higher is superior)					
	HMCS⟨1⟩	HMCS⟨2⟩	HMCS⟨3⟩	FP-HMCS⟨3⟩	FP-EH-AHMCS⟨3⟩	FP-LH-AHMCS⟨3⟩
10^0 μ sec	0.34×	0.34×	0.34×	0.32×	0.34×	0.37×
10^1 μ sec	0.14×	0.37×	0.47×	0.68×	0.73×	0.68×
10^2 μ sec	0.10×	0.34×	0.47×	0.77×	0.70×	0.71×
10^3 μ sec	0.12×	0.31×	0.47×	0.78×	0.74×	0.70×
10^4 μ sec	0.14×	0.32×	0.42×	0.78×	0.73×	0.69×
10^5 μ sec	0.12×	0.28×	0.47×	0.77×	0.72×	0.68×

Table 4.17 : Worst-case fractional throughput of various locks (compared to the best performing lock at the same contention level) over a range of contention levels varying for different periods. Critical section size = 4 cache line updates.



(a) Mean fractional throughput

(b) Worst-case fractional throughput

Figure 4.34 : Relative throughput of various locks (compared to the best performing HMCS lock) at different contention levels for different periods of contention. Critical section size = 4 cache line updates.

From these statistics, we infer that one microsecond is the lower bound on how fast an FP-AHMCS lock can react to the changing contention on Power 755. For cases where the rate of change of contention is larger than one microsecond, we make the following key observations from these statistics:

1. The *mean fractional throughput* $\mathcal{M}$ of FP-EH-AHMCS$\langle 3 \rangle$ and FP-LH-AHMCS$\langle 3 \rangle$ locks for short critical sections (2 cache lines) typically reaches more than 94% and 87%, respectively. This performance surpasses the *mean fractional throughput* of all static HMCS locks by a significant margin. The *mean fractional throughput* of FP-HMCS$\langle 3 \rangle$ lock for short critical sections is just below that of FP-AHMCS locks (87-89%).

2. Both FP-EH-AHMCS$\langle 3 \rangle$ and FP-LH-AHMCS$\langle 3 \rangle$ locks provide better protection against the worst-case behavior (*worst-case fractional throughput* $\mathcal{W}$) for short critical sections compared to the other locks (more than 51%). FP-EH-AHMCS$\langle 3 \rangle$ is particularly robust (more than 80% $\mathcal{W}$) since it quickly adapts to changing contention when compared to FP-LH-AHMCS$\langle 3 \rangle$.

3. HMCS$\langle 3 \rangle$, HMCS$\langle 2 \rangle$, and HMCS$\langle 1 \rangle$, in that order, degrade in their *mean fractional throughput* as well as *worst-case fractional throughput*.

4. As the size of the critical section increases (4 cache lines), the HMCS$\langle 3 \rangle$ catches up with the adaptive locks in terms of its *mean fractional throughput*. This behavior is natural since the cost of acquiring additional locks does not appear on the critical path. For the same reason the FP-HMCS$\langle 3 \rangle$ starts to outperform the FP-AHMCS$\langle 3 \rangle$ locks, in terms of both *mean fractional throughput* and *worst-case fractional throughput*. The FP-AHMCS locks are, however, of significance when the memory hierarchy becomes deeper than three levels.

Overall, the combination of hysteresis, fast-path, and HMCS lock shows superior performance over a static-depth HMCS lock.

4.11.5 Evaluation of HTM on IBM POWER8

We evaluated the AHMCS lock with hardware transactional memory (HTM) on a machine with two IBM POWER8 processors. Each processor has 12 8-way SMT cores clocked at 2 GHz. There is a total of 192 hardware threads on the machine. Each core has 32 KB 4-way L1 cache, 512 KB 8-way L2 cache, and 8 MB L3 8-way cache. The total L3 cache on each processor is 96 MB. The processor uses a Non-Uniform Cache Architecture (NUCA) cache policy [115], where the hot lines migrate within the caches.

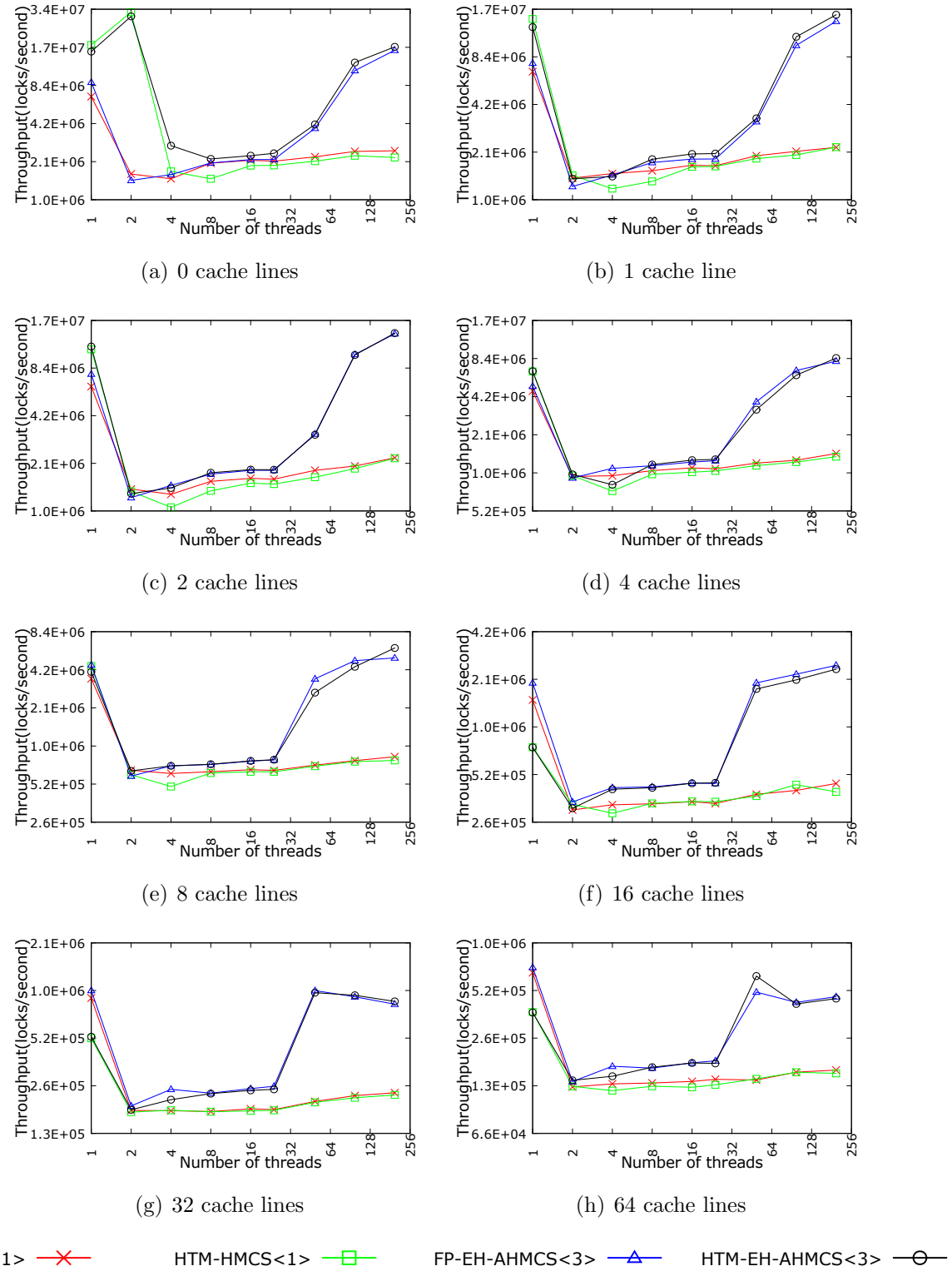
Figure 4.35 compares the performance of the following four different mutual exclusion implementations:

1. **HTM-HMCS**$\langle 1 \rangle$: HTM with an HMCS$\langle 1 \rangle$ lock as its fallback,
2. **HMCS**$\langle 1 \rangle$,
3. **HTM-EH-AHMCS**$\langle 3 \rangle$: HTM with an EH-AHMCS$\langle 3 \rangle$ lock as its fallback, and
4. **FP-EH-AHMCS**$\langle 3 \rangle$.

HTM-HMCS$\langle 1 \rangle$ vs. HMCS$\langle 1 \rangle$ is a valuable comparison of hardware-based vs. software-based synchronization, respectively. These two locks should perform well under low contention; both should degrade in performance under high contention.

Similarly, HTM-EH-AHMCS$\langle 3 \rangle$ vs. FP-EH-AHMCS$\langle 3 \rangle$ is a valuable comparison of hardware-based vs. software-based synchronization. These two locks should perform well under a variety of contention levels.

We evaluate the implementations for different levels of contention and different critical section sizes. We give 30 seconds for each experimental run. We have evaluated several other HMCS variants presented in this chapter by augmenting them with HTM, however, for brevity, we do not discuss details of each one of them.



(a) 0 cache lines

(b) 1 cache line

(c) 2 cache lines

(d) 4 cache lines

(e) 8 cache lines

(f) 16 cache lines

(g) 32 cache lines

(h) 64 cache lines

Figure 4.35 : Comparison of hardware-based locks (HTM-HMCS⟨1⟩ and HTM-EH-AHMCS⟨3⟩ locks) and software-based locks (HMCS⟨1⟩ and FP-EH-AHMCS⟨3⟩) on IBM POWER8.

From Figure 4.35, we make the following key observations:

1. For empty critical sections, the HTM implementations clearly win over all other locks. However, this scenario has little if any practical application.

2. When contention is low and critical sections are short (1-4 cache lines), the use of HTM improves the throughput compared to their counterparts that do not use HTM.

3. When critical section sizes are large, using HTM has detrimental effects compared to their non-HTM counterparts even when there is no contention. This effect results from more frequent transaction failures for large transactions due to the hardware imposed transactional capacity.

4. Finally, HTM is not a panacea for synchronization. Sophisticated locks, such as FP-AHMCS, are needed even if HTM becomes more pervasive. In general, hardware transactions that fall back to an HMCS$\langle 1 \rangle$ lock yield much lower performance than an FP-EH-AHMCS$\langle 3 \rangle$. Furthermore, hardware transactions that fall back to an EH-AHMCS$\langle 3 \rangle$ lock yield negligible overall performance improvement over an FP-EH-AHMCS$\langle 3 \rangle$, which shows the efficacy of our software-based solutions.

4.12 Discussion

In this chapter, we demonstrated the need for better mutual exclusion techniques for emerging architectures that employ a Non-Uniform Memory Access (NUMA) design. The HMCS lock design mirrored the underlying hardware's NUMA hierarchy in software, which enabled it to deliver a high throughput under high contention. Since C-MCS-MCS lock protocol typically passes a lock at the socket level, it incurs higher lock passing latency compared to an HMCS protocol, which typically passes a lock at the core level. For that reason, the HMCS lock has lower passing latency and superior throughput compared to a C-MCS-MCS lock. By retrofitting the C-MCS-MCS lock, which formed cohorts among cores sharing a socket, to instead form cohorts among threads sharing a core, we offered an alternative cohort lock that can match the throughput of an HMCS lock under extreme contention. This approach differs from the design point that Dice et al. intended for their C-MCS-MCS lock. While

the C-MCS-MCS lock with core-level cohorting matches the throughput of an HMCS lock, it requires a passing threshold that yields much greater unfairness. Analytically modeling these sophisticated locks' throughput and fairness enabled us to compare and contrast their properties.

While a fixed-depth HMCS lock showed high performance under high contention, it was not ideal under low contention. The FP-AHMCS design enhanced the HMCS lock by adding a software-based fast-path and hysteresis for dynamically adjusting to any contention level. This adaptation made FP-AHMCS a set of versatile locks, which are superior to fixed-depth hierarchical queuing locks.

Incorporating the speculative mutual exclusion feature offered by Hardware Transactional Memory (HTM) atop an AHMCS lock further enhanced our lock design. The HTM augmented AHMCS ensures low latency under low contention by managing mutual exclusion in hardware and falls back to software for managing high contention. The design not only ensures correct interplay between hardware and software methods but also puts each of their strengths to use—hardware for low latency under low contention and software for high throughput under high contention. Our software-based contention management carefully choreographs how locks are passed from one requester to the other by prioritizing locality without completely sacrificing fairness. The guarantees provided by our software-based fast-path mechanism are invaluable since they offer a low latency that even an HMT-based synchronization fails to match in some situations. As systems with many cores and deep, distributed NUMA hierarchies become more pervasive, our hardware-software cooperative design with dynamic adaptation shows promise for efficient synchronization over a broad range of contention levels.

Chapter 5

Identifying Unnecessary Memory Accesses

Time moves in one direction, memory in another.

William Gibson

Once a parallel program is load balanced and its parallel overheads are eliminated, the next step to improve its performance is to tune individual threads of sequential execution. Exhaustive coverage of tuning a thread of sequential execution is outside the scope of this dissertation. In this chapter, we address one particular, albeit important, aspect of tuning individual threads of sequential execution—the overhead of unnecessary memory accesses.

5.1 Motivation and overview

On modern architectures, memory accesses are costly. For many programs, exposed memory latency accounts for a significant fraction of execution time. Unnecessary memory accesses, whether cache hits or misses, lead to poor resource utilization and have a high energy cost as well [89]. In the era where available memory bandwidth per processor core is shrinking [102, 153], gratuitous memory accesses cause performance losses.

A *dead write* occurs when two successive writes happen to a memory location without an intervening read of the same location. Dead writes are useless operations. Traditional performance analysis tools lack the ability to detect inefficiencies related to dead writes. Compiler optimizations that reduce memory accesses by register allocation of scalars (e.g., [44]) or array elements (e.g., [35]) are critical to high performance. Other optimizations that eliminate redundant computations [21, 55, 117] are similarly important. However, none of these optimizations is effective in the global elimination of dead writes. Compile-time elimination of all dead writes is complicated by issues such as aliasing, aggregate types,

optimization scope, late binding, and path- and context- sensitivity.

Dead writes are surprisingly frequent in complex programs. To pinpoint dead writes, we developed DeadSpy—a tool that monitors every memory access, identifies dead writes, and provides actionable feedback to guide application tuning. Using DeadSpy to analyze the reference executions of the SPEC CPU2006 benchmarks [85] showed that the integer benchmarks had over 15% dead writes and the floating-point benchmarks had over 6% dead writes. On some inputs, the SPEC CPU2006 `403.gcc` benchmark had as many as 76.2% dead writes.

Besides providing a quantitative metric of dead writes, DeadSpy reports source lines and complete calling contexts involved in high frequency dead writes. Such quantitative attribution pinpoints opportunities where source code changes could significantly improve program efficiency. Various causes of inefficiency manifest themselves as dead writes at runtime. We show cases where lack of or inefficient compiler optimizations cause dead writes; we also highlight cases where the programmer did not design for performance. We restructure codes that have significant fractions of dead writes and demonstrate performance improvements. To the best of our knowledge, DeadSpy is the first dynamic dead write detection tool. The proposed methodology of eliminating dead writes is a *low-cost high-yield* strategy when looking for opportunities to improve application performance.

We restructure codes that have significant fractions of dead writes and demonstrate performance improvements. Insights provided by DeadSpy helped significantly improve the performance of several important code bases such as `Chombo` [11], `NWChem` [230], `bzip2` [206], `gcc` [85], and `hmmer` [85]

5.2 Contributions

This work, part of which appeared in the Proceedings of the 10^{th} International Symposium on Code Generation and Optimization [42], makes the following contributions.

1. It identifies dead writes as a symptom of inefficiency arising from many causes,

2. It proposes elimination of dead writes as an opportunity for improving performance,

3. It presents a tool to count dead writes in an execution and precisely attributes these counts to source lines along with calling contexts,

4. It analyzes a variety of benchmark programs and shows that the fraction of dead writes is surprisingly high,

5. It identifies several cases where dead writes result from ineffective compiler optimizations, and

6. It demonstrates through several case studies, how eliminating causes of dead writes significantly improves performance.

5.3 Chapter roadmap

The rest of this chapter is organized as follows. Section 5.4 presents a new methodology for detecting program inefficiencies via tracking dead writes. Section 5.5 sketches the design and implementation of DeadSpy. Section 5.6 evaluates benchmark programs using DeadSpy. Section 5.7 studies five codes to explore the causes of dead writes and the performance benefits of dead write elimination. Finally, Section 5.8 ends with some discussion.

5.4 Methodology

In this section, we describe a dynamic dead write detection algorithm. The driving principle behind our tool is the invariant that *two writes to the same memory location without an intervening read operation make the first write to that memory location dead.*

To identify dead writes throughout an execution, DeadSpy monitors every memory read and write operation issued during program execution. For each addressable unit of memory, DeadSpy maintains the most recent access history. The access history takes one of the following values:

Virgin `(V)`**:** the memory address has never been accessed,

Read `(R)`**:** the last access was a read operation, or

Written `(W)`**:** the last access was a write operation.

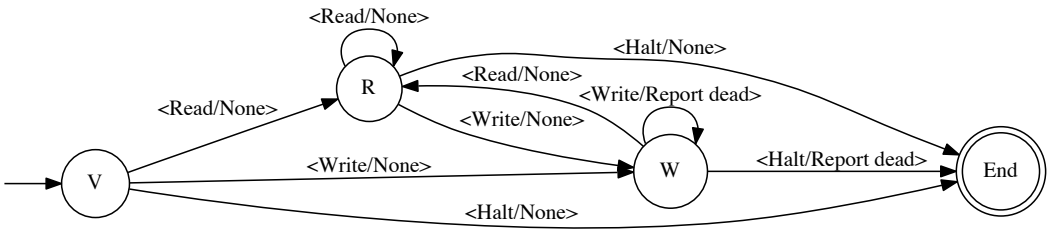


Figure 5.1 : State transition diagram.

The state transitions of each memory location obey the automaton shown in Figure 5.1, where the transition edges are labeled with `<instruction/action>` pairs. Every memory location `M` starts in the *Virgin* state. An instruction that reads `M` updates `M`'s state to `R`. An instruction that writes `M` updates `M`'s state to `W`. A state transition from `W` to `W` corresponds to a dead write. When a dead write is detected, some additional information is recorded for later reporting. Similarly, a final write to `M` without a subsequent read qualifies as a dead write. All other state transitions have no actions associated with them. A `halt` instruction transitions the automaton to the terminating state. When the program terminates, DeadSpy also terminates. Since the automaton considers the effect of each memory operation executed in the program from start to finish, there can be no false-positives or false-negatives. The approach is sound even in the event of asynchronous control transfers.

For multi-threaded programs, if two consecutive writes to a location are from two different threads, this may indicate a source of non-determinism or a data race, unless protected by a common lock. In this work, our focus is to identify program inefficiencies for a single thread of execution; for multi-threaded executions, we treat such writes to a memory location simply as dead writes.

5.5 Design and implementation

We implemented DeadSpy using Intel's dynamic binary instrumentation tool *Pin* [144] to monitor every read and write operation. We use *shadow memory* [167] to maintain the access

history of each memory location. We instrument each function's `CALL` and `RETURN` instructions to build a dynamic *calling context tree (CCT)* [9]. A CCT enables us to compactly store the program contexts necessary for reporting dead writes. Each interior node in our CCT represents a function invocation, and each leaf node represents a write instruction. We apportion each instance of redundancy to a pair of call paths involved in dead writes. At program termination, we present the dead writes observed for the entire execution. In our implementation, we decided not track the dead writes happening during the final `W` $\rightarrow$ `End` transitions. Typically, such dead writes have a negligible contribution to the overall dead writes in a program. Furthermore, such dead writes may be infeasible to eliminate in practice.

In the following subsections, we first introduce the terminology used in the rest of this chapter and briefly describe Pin. We then describe our implementation of DeadSpy, including its use of shadow memory and CCT construction, along with its strategies for recording and reporting dead writes. Next, we present the challenges involved in attributing dead writes to source lines and then we sketch the details of our solution. We conclude this section with the details of accounting and attributing dead writes.

5.5.1 Terminology

In the context of this chapter, we define the following terms:

- *Read* is an instruction that has the side effect of loading a value from memory.
- *Write* is an instruction that has the side effect of storing a value into memory.
- *Operation* represents a dynamic instance of an instruction.

For a `W` $\rightarrow$ `W` state transition at location `M`, we define:

- *Dead context* as the program context in which the first write happened; this context wrote an unread value and hence is a candidate for optimization.
- *Killing context* as the program context that overwrote the previously written location without an intervening read of the location.

5.5.2 Introduction to Pin

Pin is a dynamic binary instrumentation tool. Pin provides a rich set of APIs to inject instrument (*analysis routines*) into a program at different granularities such as module, function, trace, basic block, and instruction. A *trace*, in the Pin jargon, is a single entry multiple exit code sequence—for example, a branch starts a new trace at the target, and a function call, return, or a jump ends the trace. Instrumentation occurs immediately before a code sequence is executed for the first time. Using Pin, we instrument every read and write instruction to update the state of each byte of memory affected by the operation. Pin also provides APIs to intercept system calls, which we use to update the shadow memory to account for side effects of system calls.

5.5.3 Maintaining memory state information

In our implementation, we maintain the current state of each memory location in a *shadow memory*, analogous to *Memcheck*—a Valgrind tool [167]. Each memory byte `M` has a shadow byte to hold its previous access history represented as `STATE(M)`. The shadow byte of `M` can be accessed by using `M`'s address to index a two-level page table. We create chunks of 64KB shadow memory pages on demand. On 64-bit x86_64 machines, only the lower 48 bits are used for the virtual address mapping [8]. With 64KB shadow pages, the lower 16 bits of address provide an offset into a shadow page where the memory status is stored. The higher 20 bits of 48 bits, provide an offset into the first-level page table, which holds 2^{20} entries—on a 64-bit machine it occupies 8MB space. The middle 12 bits provide an offset into a second-level page table. Each second-level page table has 2^{12} entries and occupies 32KB on a 64-bit machine. Each second-level page table is allocated on demand *iff* an address is accessed in its range. We adopt a *write-allocate* semantics so that read-only pages do not get shadowed. To maintain the context for a write operation, we store an additional pointer sized variable `CONTEXT(M)` in the shadow memory for each memory byte `M`. Thus, we have one shadow byte of state and an 8-byte context pointer for a total of nine bytes of metadata per data byte. Figure 5.2 shows the organization of shadow memory; numbers in dark circles represent the steps involved in address translation.

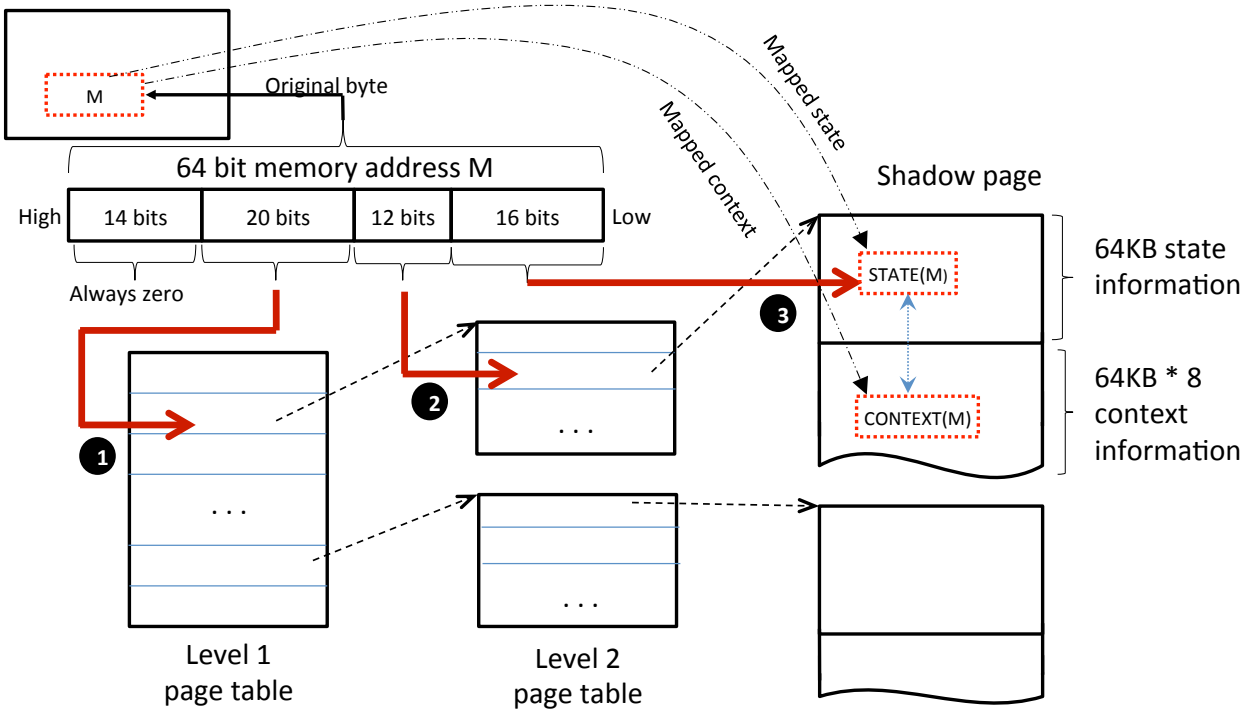


Figure 5.2 : Shadow memory and address translation.

5.5.4 Maintaining context information

To accurately report the *dead* and *killing contexts* involved in every dead write, each instruction writing to location `M` also updates `CONTEXT(M)` and records a cursor to the call chain needed to recover the calling context and the instruction pointer (IP) of the write. We accomplish this by maintaining a CCT. The CCT dynamically grows as the execution unfolds. In this subsection, we provide the details of building a simple CCT; additional details of including the IP information to map back to source lines are presented in Section 5.5.7.

To build a simple CCT, we insert instrumentation before each `CALL` and `RETURN` machine instruction. DeadSpy maintains a cursor (`curCtxtNodePtr`) that points to a CCT node representing the current function. The path from the `curCtxtNodePtr` to the CCT root represents the current call stack. The analysis routine executed before each `CALL` instruction creates a new `ContextNode` for the callee under the `curCtxtNodePtr` if not already present, and updates the `curCtxtNodePtr` to point to the callee `ContextNode`. The analysis routine executed before each `RETURN` instruction updates the `curCtxtNodePtr` to its parent

`ContextNode`. The analysis routine executed just before each write instruction updates the pointer-sized location `CONTEXT(M)` in the shadow memory with the location pointed to by the `curCtxtNodePtr`. If the write operation is multi byte, we update `CONTEXT(M)` for each of the data bytes written by the instruction with the value of the `curCtxtNodePtr`. For multithreaded codes, there will be one CCT per thread; we enforce atomicity of the execution of an instruction and its analysis routine.

5.5.5 Recording dead writes

When a `W` $\rightarrow$ `W` state transition is detected, we record a 3-tuple <*dead context pointer, killing context pointer, frequency*> into a table—`DeadTable`. Each entry in the `DeadTable` is uniquely identified by <*dead context pointer, killing context pointer*> ordered pair. For example, `<CONTEXT(M), curCtxtNodePtr,1>` is the 3-tuple that DeadSpy might insert into the `DeadTable` when a dead write is observed for a location `M`. If such a record is already present, its *frequency* is incremented. The *frequency* accumulates the number of bytes dead for a given pair of contexts. We discuss the details of accounting and attributing dead writes in Section 5.5.8.

5.5.6 Reporting dead and killing contexts

At program termination, all 3-tuples are retrieved from the `DeadTable` and sorted by decreasing *frequency* of each record. For each tuple, the full call chain representing its *dead context* is obtained by traversing parent links in the CCT starting from the *dead context pointer*[1]. Similarly, we can retrieve the *killing context* by traversing parent links in the CCT starting from the *killing context pointer*.

5.5.7 Attributing to source lines

Sometimes having only a chain of function names as a context may not suffice; source line numbers involved in the dead and killing writes may be needed. In the following paragraphs, we present challenges involved in making line-level attribution along with our solutions.

[1]For library calls where the target of a call is to a trampoline, we disassemble target address of the jump to extract the correct function name.

5.5.7.1 Overhead of a naive approach

Attributing to source or instruction, in addition to the function, requires recording the instruction pointer (IP) in addition to the enclosing function for each write operation. Naively recording additional pointer sized IP values bloats shadow memory and adds excessive overhead. To avoid the space overhead, one could consider adding write IPs as leaf nodes in the CCT; these nodes would represent writes within a parent function. However, every time a write operation is executed, recording the context would involve looking up the corresponding CCT node for the write IP; this would dramatically inflate the time overhead of monitoring.

5.5.7.2 Strategy to capture the calling context with instruction pointer

Instead of recording each IP as a pointer in the shadow space, one could consider representing the set of write instructions in a function as an array and assigning a slot index to each. Using Pin, one can assign a unique slot index to each write instruction during JIT translation; with this approach, the slot index for each write instruction is available to an analysis routine during execution in constant time. This approach would address the aforementioned problem of lookup overhead. Pin makes the information about function boundaries available to the instrumentation at run time. This information, in theory, could be used to scan each function and assign a slot index to each write instruction at the time of injecting the instrumentation into an application's binary code.

5.5.7.3 Challenges due to imprecise binary analysis

A problem with the aforementioned approach of scanning a function and associating store instructions in a function with slot indices requires precise knowledge of function boundaries as well as precise disassembly of function. In practice, program disassembly is imprecise [204] and discovering function boundaries relies upon compiler-generated symbol information, which is often incomplete and sometimes incorrect [224]. The approach of associating write instructions in a function with slots would fail when execution transfers to an arbitrary code region where precise function bounds are unavailable. This behavior is not

uncommon; even the startup code executed before reaching the `main()` function exercises such corner cases.

Providing a perfect solution to track IPs of write operations involves complex engineering of the CCT. To facilitate this, we developed a strategy that makes use of *traces* that Pin generates at JIT time. While the function bounds can be incorrect, the instructions identified in a trace are always correct since trace extraction happens at runtime. We modify the aforementioned simple CCT such that each `ContextNode` has several child traces (`ChildTraces`), each one represented by a `TraceNode`. Each `TraceNode` has an array of child IPs `(ChildWriteIPs)`, where each array element corresponds to a write instruction under that trace, as shown in Figure 5.3. The content of each slot of a `ChildWriteIPs` array is simply a pointer to its parent `TraceNode`. This organization allows us to obtain a pointer to a leaf-level `ChildWriteIPs` slot and traverse a chain of pointers from leaf to the root to recover an entire call chain along with the encoded leaf-level instruction pointer.

5.5.7.4 Pin's trace instrumentation to obtain precise disassembly

Since a leaf-level slot does not contain the actual IP, one has to decode it (typically, during post processing to obtain the call-path in a human readable form). We maintain a reverse map (`RMap`) from each Pin trace to its constituent write IPs. Specifically, the i^{th} slot in `ChildWriteIPs` of a trace T with unique identifier `traceId`, is the IP recorded at `RMap[traceId][i]`. This reverse mapping allows us to recover the IP given the index of a write instruction in a `ChildWriteIPs`. Note that we do not need to maintain a reverse mapping from each `ChildWriteIPs` to the constituent write IPs since a pin trace may appear as different `ChildWriteIPs` at different calling contexts.

We intercept each Pin trace creation assigning a unique identifier (`traceId`) for each trace. On each trace creation, we also walk the instructions belonging to a trace and populate the `RMap` with the key as the `traceId` and the value as an array of write instructions belonging to the trace.

We instrument each trace entry. We also add instrumentation before each `CALL` instruction to set a flag. On entering a trace, if the flag is set, we know that we have just entered a new function. If the flag is set when entering a trace, we inspect the children of the current

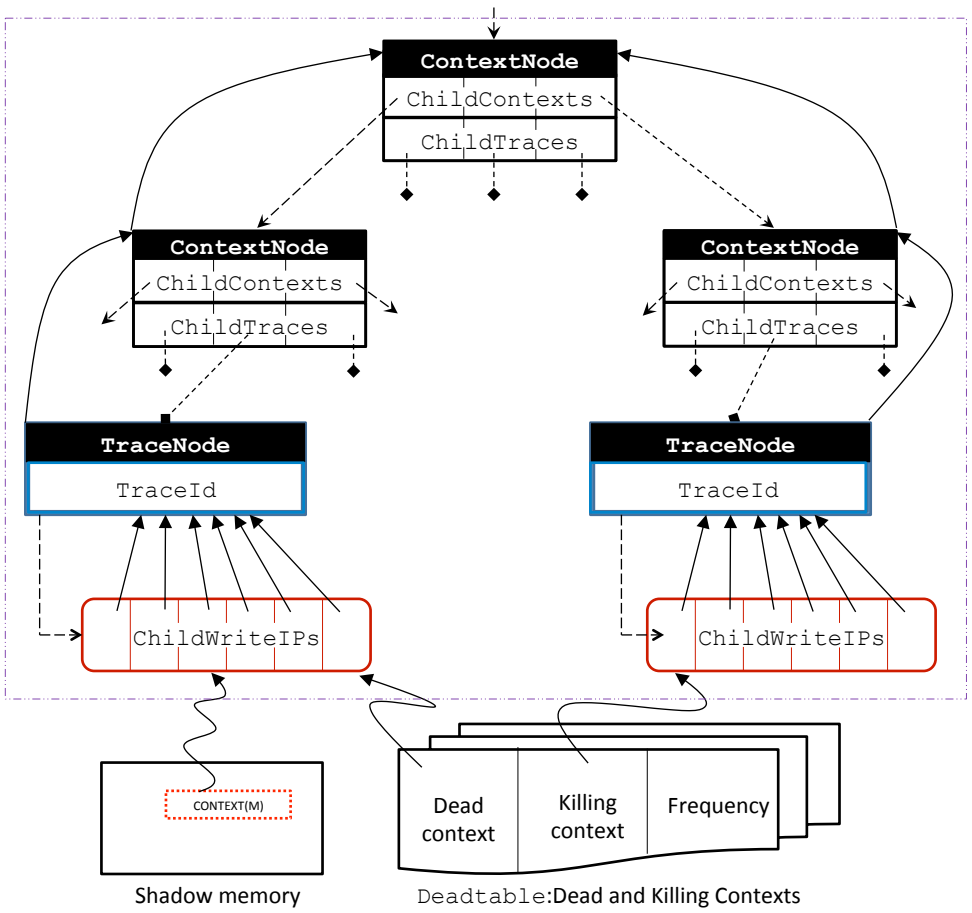


Figure 5.3 : Calling context tree.

CCT node `curCtxtNodePtr` looking for a child `ContextNode` representing the current IP; if none exists, we create and insert one. We update the `curCtxtNodePtr` to the appropriate child and reset the flag. Whether the flag was set or not, we next look up a child `TraceNode` with the id of the current trace (`traceId`) under `curCtxtNodePtr` (creating and inserting a new one if necessary), and set `curTraceNodePtr` to point to the current trace. *Tail calls* to known functions are handled by having Pin instrument function prologues to adjust the `curCtxtNodePtr` to point to a sibling node followed by adjusting the `curTraceNodePtr`. A tail call to an instruction sequence not known to be a function ends up being associated with a trace under the current function. Finally, we add instrumentation before each write instruction to update a pointer `(curChildIPIndex)` to point to a slot of the `ChildWriteIPs` that is currently being executed.

Encoding and decoding the call-path: Using the aforementioned CCT structure, let's consider how we maintain the shadow information for a write operation. If a write instruction `W`$_i$ is numbered as the 42^{nd} write while JIT-ing a trace `T`; then executing `W`$_i$, which writes to location `M`, would update `CONTEXT(M)` with `&(T->ChildWriteIPs[42])`. Note that at the time of recording this information, `curTraceNodePtr` would be pointing to `T` and `curChildIPIndex` would be 42. If an instance of `W`$_i$ is a killing write, before updating `CONTEXT(M)`, we record or update the 3-tuple `<CONTEXT(M), &T->ChildWriteIPs[42], frequency>` in the `DeadTable`.

During the post processing, when we want to recover the full call-path, we will only have a pointer, say `p`, to `ChildWriteIPs[42]`. Dereferencing the pointer `p` provides us a pointer to the `TraceNode T` encapsulating the `ChildWriteIPs[42]`. The pointer arithmetic, `(p - T->ChildWriteIPs)`, yields the index of the slot that `p` points to; in this case 42. Now, we can recover the real IP by indexing `RMap[T->traceId][42]`. To recover the rest of the call-path, we can follow the parent links starting at `T->parent` till the root of the tree.

We note that the method developed for attributing to the calling context reports the source line for the leaf level of the dead and killing writes but does not provide the source level information for interior nodes of the call-path. The interior node will contain only function names. Chapter 6, which focuses on call-path attribution, not only eliminates this deficiency but also reduces the overhead of call-path attribution.

5.5.8 Accounting dead writes

Read and write operations happen at different byte-level granularities, for example 1, 2, 4, 8, 16, etc.. We define $\overline{AverageWriteSize}$ in an execution as the ratio of the total number of bytes written to the total number of write operations performed.

$$\overline{AverageWriteSize} = \frac{NumBytesWritten}{NumWriteOps}$$

We approximate the number of dead write operations in an execution as the ratio of the total number of bytes dead to its $\overline{AverageWriteSize}$.

$$\widehat{NumDeadOps} = \frac{NumBytesDead}{\overline{AverageWriteSize}}$$

We define *Deadness* in an execution as the percentage of the total dead write operations out of the total write operations performed, which is same as the percentage of the total number of bytes dead out of the total number of bytes written.

$$\begin{aligned} Deadness &= \frac{\widehat{NumDeadOps}}{NumWriteOps} \times 100 \\ &= \frac{NumBytesDead}{NumBytesWritten} \times 100 \end{aligned}$$

DeadSpy accumulates the total bytes written and the total operations performed in an execution. As stated in Section 5.5.5, each record C_{ij} in a `DeadTable` is associated with a *frequency* $F(C_{ij})$, representing the total bytes dead in a *dead context* i due to a *killing context* j. We compute total *Deadness* as:

$$Deadness = \frac{\sum_i \sum_j F(C_{ij})}{NumBytesWritten} \times 100$$

We apportion *Deadness* among contributing pair of contexts C_{pq} as:

$$Deadness(C_{pq}) = \frac{F(C_{pq})}{\sum_i \sum_j F(C_{ij})} \times 100$$

An alternative metric to *Deadness* is *Killness* — the ratio of the total write operations killing previously written values to the total write operations in an execution. In practice, we found that both *Deadness* and *Killness* values for programs are almost the same.

Deadness is insensitive to the locality of reference. Successive dead writes to different memory locations that share a cache line (spatial locality) may not incur a significant memory access latency. Furthermore, write buffering can alleviate the latency of dead writes. One may devise a metric of dead writes that simulates the effects of caches and write buffers

analogous to [135]. Dead writes are wasteful operations in any case that consume hardware resources. Dead writes can aggravate capacity misses and also cause frequent flushing of the write buffer. We found the deadness metric to be quite reliable in pinpointing higher-level problems such as poor data structure choice that are not tightly coupled to an underlying hardware design.

5.6 Experimental evaluation

Experimental setup. For most of our experiments, we used a quad-socket system with four `AMD Opteron 6168` processors clocked at `1.9 GHz` with 128GB of 1333MHz DDR3 running `CentOS 5.5`. We used the `GNU 4.1.2` [74] compiler tool chain with `-O2` optimization. For the `NWChem` case study, we used a different machine, and we provide its details later.

5.6.1 SPEC benchmarks

5.6.1.1 Deadness in SPEC CPU2006

We measured the deadness of each of the SPEC CPU2006 integer and floating-point reference benchmarks, and the results are shown in Figures 5.4 and 5.5 respectively. Several benchmarks execute multiple times, each time with a different input; each column in Figures 5.4 and 5.5 shows the measurements averaged over different inputs for the same benchmark. For a benchmark with multiple inputs, error bars represent the lowest and the highest deadness observed on its different inputs. The average deadness for integer benchmarks is 15.1% with the highest of 76.2% for `gcc` on the input `c-typeck.i`, and the lowest of 3.2% for `astar` on the input `rivers.cfg`. The average deadness for floating-point benchmarks is 6.5% with the highest of 33.9% for `soplex` on the input `pds-50.mps`, and the lowest of 0.3% for `lbm`. The average difference between *Killness* and *Deadness* is 1% for integer benchmarks and 0.07% for floating-point benchmarks.

These measurements indicate that for several of these codes, a large fraction of memory access operations is dead. Often, just a few pairs of contexts account for most of dead writes. For example, in the SPEC CPU2006 integer reference benchmarks, for the benchmark/input pair with the median deadness, its top five context pairs account for 90% of deadness, and

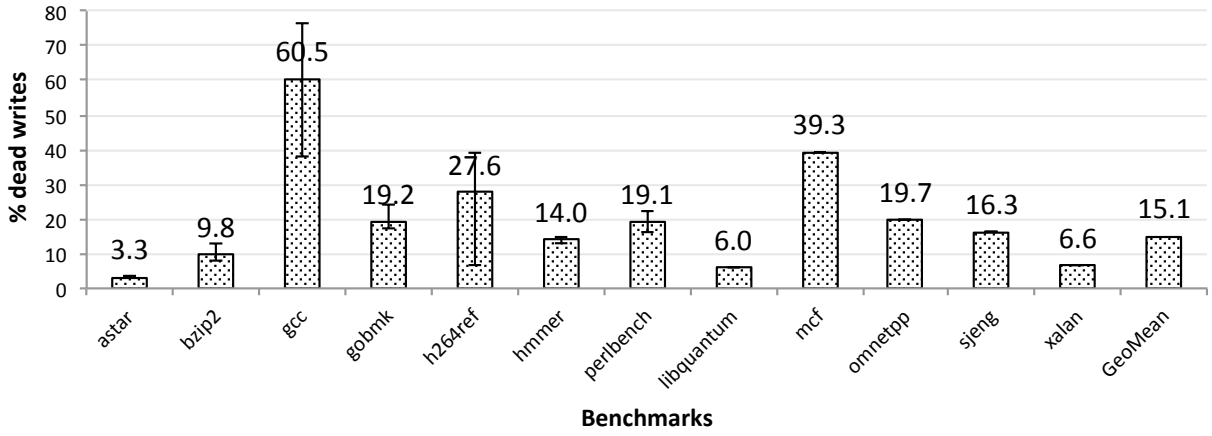


Figure 5.4 : Dead writes in SPEC CPU2006 integer reference benchmarks.

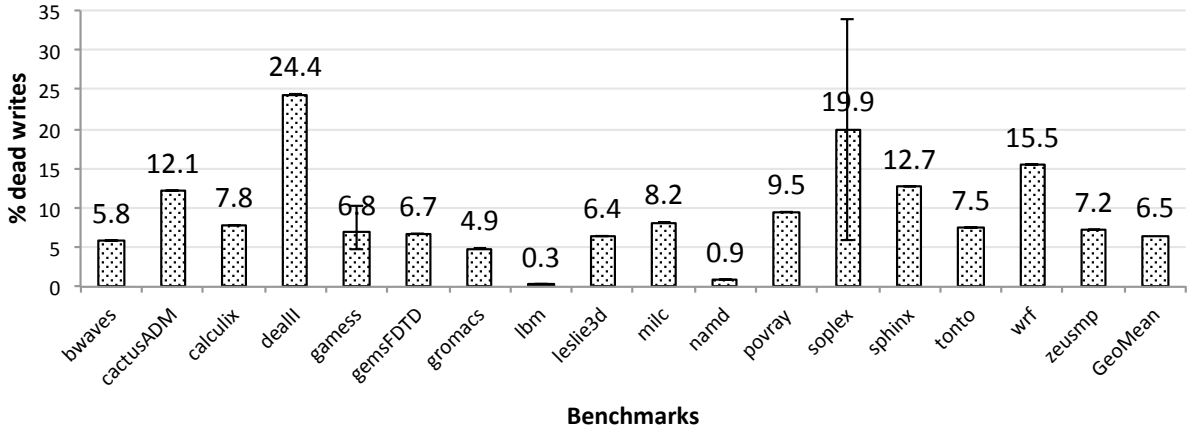


Figure 5.5 : Dead writes in SPEC CPU2006 floating-point reference benchmarks.

its top 15 context pairs account for 95% of deadness. This phenomenon indicates that a domain expert could optimize a handful of context pairs presented by DeadSpy and expect to eliminate most dead writes in a program.

5.6.1.2 Overhead of instrumentation

As a tool that monitors every read and write operation in a program, quite naturally, DeadSpy significantly increases execution time. Figure 5.6 shows the overhead of each of SPEC

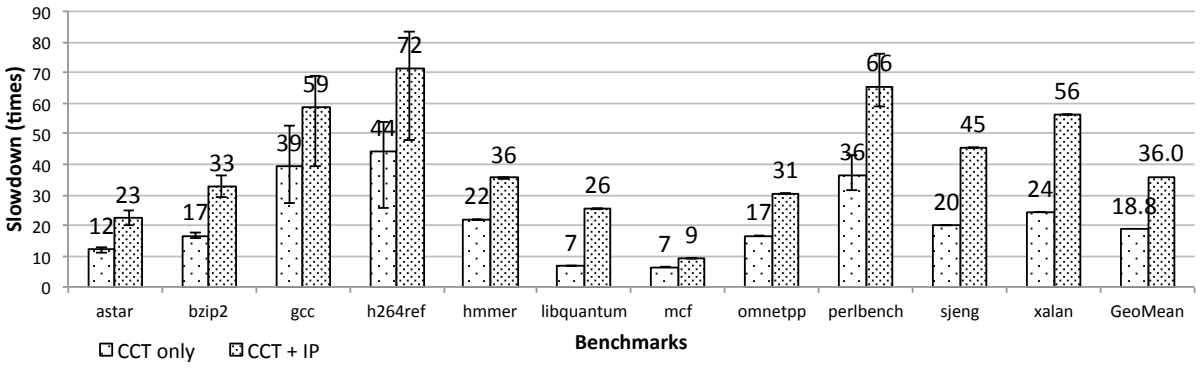


Figure 5.6 : DeadSpy overhead for SPEC CPU2006-INT.

CPU2006 integer reference benchmarks. The overhead to obtain dead and killing contexts without line numbers is much less than when line number information is tracked as well. To track line numbers, we need to instrument each trace entry. The average slowdowns due to instrumentation without and with line information are 18.8x and 36x, respectively. We ignored the `445.gobmk` benchmark for computing the average due to its large memory footprint created by deep recursion. The maximum slowdown is seen for the `h264ref` benchmark on the input `foreman_ref_encoder_baseline.cfg`, which is 41.3x without line-level attribution and 83.6x with line-level attribution. High deadness typically comes with high overhead due to the cost of recording participant contexts on each instance of a dead write.

5.6.1.3 Deadness across compilers and optimization levels

We used `-O2` optimization as the basis for our experiments since it is often used in practice. However, our findings are not limited to a specific optimization level or a specific compiler. We found high deadness across different optimization levels and different compilers. For completeness, we present the deadness in SPEC CPU2006 integer reference benchmarks when compiled *without* optimization (`-O0`), with *default* optimization (`-O2`), and with *highest* level of optimization on three different compiler chains viz., `Intel 11.1` [96], `PGI 10.5` [226], and `GNU 4.1.2` [74]. By reading the accompanying compiler manual pages, we concluded that the highest optimization level for `Intel 11.1` is `-fast`, for `PGI 10.5` is

Program	Deadness in %								
	Intel 11.1			PGI 10.5			GNU 4.1.2		
	-O0	-O2	max	-O0	-O2	max	-O0	-O2	max
astar	2.3	8.4	5.0	5.2	1.1	5.7	2.5	3.3	7.7
bzip2	4.7	8.6	8.9	4.9	9.9	12.8	5.1	9.8	11.9
gcc	39.3	67.8	67.2	40.8	53.3	51.9	39.7	60.5	64.5
gobmk	15.7	21.3	22.7	17.4	19.1	20.7	16.0	19.2	20.1
h264ref	14.4	28.4	38.3	15.1	24.5	26.2	15.3	27.6	27.9
hmmer	31.3	68.7	68.8	31.5	67.6	67.9	0.3	14.0	29.4
perlbench	15.3	18.0	20.0	16.7	16.5	16.8	13.2	19.1	n/a
libquantum	1.7	6.0	0.3	2.8	7.1	7.5	2.3	6.0	6.1
mcf	16.6	49.4	49.5	17.2	27.6	47.2	17.3	39.3	47.3
omnetpp	4.8	19.4	22.1	11.8	11.2	27.7	4.7	19.7	21.4
sjeng	9.6	20.4	17.8	10.3	11.4	13.9	10.0	16.3	19.7
xalan	1.5	6.6	6.7	5.1	4.7	9.4	1.7	6.6	8.4
GeoMean	**8.2**	**19.4**	**15.3**	**11.3**	**13.4**	**19.5**	**5.9**	**15.1**	**18.7**

Table 5.1 : Deadness in SPEC CPU2006-INT with different compilers and optimization levels.

`-fastsse, -Mipa=fast,inline`, and for `GNU 4.1.2` is `-O3 -mtune=opteron`. We did not conduct profile-guided optimizations in our experiments. The deadness found across these compilers is shown in Table 5.1. The `perlbench` benchmark did not finish execution when compiled with `GNU 4.1.2` at the highest optimization, even without DeadSpy attached; hence, we do not have the results for the same. It is evident that high deadness is pervasive across compilers and across optimization levels. Intuitively, higher optimization levels, except inter-procedural analysis, offer no advantage in eliminating dead writes. Meticulous readers may observe that typically, deadness increases with increase in optimization levels on all compilers; this can be attributed to the fact that with higher optimizations, the absolute number of memory operations often reduces, but the absolute number of dead writes does not reduce proportionally.

5.6.2 OpenMP NAS parallel benchmarks

To assess the deadness in multithreaded applications, we ran DeadSpy on the OpenMP suite of NAS parallel benchmarks version 3.3 [106] with four worker threads. Table 5.2 shows the deadness found in these benchmarks. The average deadness for these applications was 3.96%, which is less than those observed for the SPEC CPU2006 serial codes. Moreover, the inter-thread deadness, which happens when a previous write by one thread is overwritten by a different thread, is negligible. For multithreaded applications tuned to maintain affinity between data and threads, there is little inter-thread deadness.

Program	Deadness in %		
	Inter-thread (A)	Intra-thread (B)	Total (A+B)
bt.S	5.28E-02	1.13E+01	1.14E+01
cg.S	6.83E-03	2.86E+00	2.87E+00
dc.S	2.26E-03	1.78E+01	1.78E+01
ep.S	9.84E-02	1.68E-01	2.66E-01
ft.S	3.35E-01	4.17E+00	4.50E+00
is.S	1.08E+00	1.06E+00	2.13E+00
lu.S	1.89E-02	6.01E+00	6.03E+00
mg.S	4.44E-01	6.10E+00	6.55E+00
sp.S	2.68E-01	3.32E+00	3.59E+00
ua.S	1.04E-01	4.43E+00	4.54E+00
GeoMean	**7.64E-02**	**3.44E+00**	**3.96E+00**

Table 5.2 : Deadness in OpenMP NAS parallel benchmarks.

5.7 Case studies

In this section, we evaluate the utility of DeadSpy for pinpointing inefficiencies in executions of four codes: the `403.gcc` and `456.hmmer` programs from SPEC CPU2006 benchmarks, the `bzip2` file compression tool [206], and two scientific applications `Chombo` [11] and `NWChem` [230]. We investigate dead writes in executions of these applications. We apply optimizations to eliminate some of the most frequent dead writes. We present the performance gains achieved after code restructuring.

5.7.1 Case study: 403.gcc

`403.gcc` was our top target for investigation since it showed a significant percentage of dead writes. For the `c-typeck.i` input, which yielded 76.2% deadness, the top most pair of dead contexts accounted for 29% of the deadness. The offending code is shown in Listing 5.1. In this frequently called function, `gcc` does the following:

1. (`line 3`) allocates `last_set` as an array of 16937 elements, 8 bytes each, amounting to a total of 132KB.
2. (`lines 5-12`) iterates through each instruction belonging to the incoming argument `loop`.
3. (`lines 7-8`) if `insn` matches a pattern, calls `count_one_set()` which updates `last_set` with the last instruction that set a virtual register.
4. (`lines 10-11`) if the basic block ends, calls `memset()` to reset the entire 132KB of the

```
1  void loop_regs_scan(struct loop *loop, ...){
2            ...
3    last_set = (rtx *) xcalloc(regs->num, sizeof (rtx));
4    /* Scan the loop, recording register usage */
5    for (each instruction in loop){
6      ...
7      if(GET_CODE (PATTERN (insn)) == SET || ...)
8        count_one_set (...,last_set,...);
9      ...
10     if (end of basic block)
11       memset(last_set,0,regs->num*sizeof(rtx));
12   }
13   ...
14 }
```

Listing 5.1: Dead writes in `gcc` because of an inappropriate data structure.

`last_set` array for reuse in the next basic block of the `loop`.

The program spends a lot of time zero initializing the array `last_set`, most of which is already zero. DeadSpy detected dead writes in `memset()` with its caller as `loop_regs_scan()`. The root cause for the high amount of deadness is that the basic blocks are typically short, and the number of registers used in a block is small; `gcc` allocated a maximum size array without considering this common case. Clearly, a *dense array is a poor data structure choice to represent this sparse register set.* We gathered some statistics on the usage pattern of `last_set` using the `c-typeck.i` input and found that the median use was only `2` *unique* slots with a maximum of 34 slots set between episodes of `memset()`s. Furthermore, the median of total number of writes to *non-unique* slots of `last_set` was 2 with a maximum of 63 between two episodes of `memset()`s. We found that just 22 non-unique slots were accessed on 99.6% of occasions. As a quick fix, we maintained a side array that recorded the first 22 non-unique indices accessed. If the side array does not overflow, we can simply zero at most those 22 indices of `last_set` instead of calling `memset()` on the entire 132KB array. Rarely, when the side array overflows, we can fall back to resetting the entire `last_set` array.

A poor choice of data-structures has manifested itself as dead writes. Better permanent fixes for the aforementioned problem include

- using a sparse representation for the register set, such as splay trees, or
- using a composite representation for the register set that switches among a short sparse vector, a scalable sparse set representation such as a splay tree, and the full dense set representation.

```
1  void cselib_init (){
2            ...
3    cselib_nregs = max_reg_num ();
4    // initializes reg_values with zeros
5    VARRAY_ELT_LIST_INIT(reg_values, cselib_nregs, ...);
6    ...
7    clear_table (1);
8  }
9
10 void clear_table (int clear_all){
11   // sets all reg_values to zeros
12   for (i = 0; i < cselib_nregs; i++)
13      REG_VALUES (i) = 0;
14      ...
15 }
```

Listing 5.2: Dead reinitialization in `gcc`.

In the latter case, a competitive algorithm could switch among representations based on the number of elements in the set and the pattern of accesses.

Another dead write context was found in the `cselib_init()` function shown in Listing 5.2. Looking at the macro `VARRAY_ELT_LIST_INIT` revealed that it was allocating and zero initializing the array `reg_values`. Then, without any further reads from the array, the call to `clear_table(1)` was again resetting all elements of `reg_values` to zeros. The dead write symptom highlights losses caused by using *generic, heavyweight APIs, where slim APIs are needed*. We fixed this by simply calling a specialized version of `clear_table()` that did not initialize `reg_values`.

We identified and fixed two more top dead contexts in `403.gcc`, both related to repeated zero initialization of a dense data structure where the usage pattern was sparse. In general, finding the root causes of performance issues was straightforward once the dead and killing contexts were presented. Optimizing the top four pairs of dead contexts resulted in improving `gcc`'s running time by 28% for the `c-typeck.i` input. The average speedup across all inputs was 14.3%. Table 5.3 shows the performance improvements as percentage speedup (%Fast column), reduction in L1 data-cache misses (`L1` column), L2 data-cache misses (`L2` column), operations completed (`Ops` column), and processor cycle counts (`Cyc` column) for each of the input files of `gcc` compared to the baseline. The performance improvements come from both reduced cache miss rates and reduced operation counts. Occasionally, there are slightly more L2 misses (represented by negative numbers) which are offset by improvements in other areas. We ran the modified `gcc` on a version of the SPEC'89 `fpppp` code, which we converted

Program	Workload	%Fast	%Reduction in resource			
			L1	L2	Ops	Cyc
403.gcc	166.i	8.5	10.8	-7.4	14.4	8.7
	200.i	3.5	6.2	-0.3	3.3	3.1
	c-typeck.i	**28.1**	29.0	31.2	29.9	24.7
	cp-decl.i	15.6	14.9	-2.3	24.0	16.8
	expr.i	18.4	14.3	12.7	23.5	18.0
	expr2.i	18.7	10.6	11.8	24.3	17.4
	g23.i	10.9	9.0	10.7	15.9	10.4
	s04.i	22.8	19.2	24.1	23.2	22.2
	scilab.i	1.9	3.6	0.0	0.7	1.4
	average	**14.3**	**13.1**	**8.9**	**17.7**	**13.7**
456.hmmer	retro.hmm	15.1	2.3	1.2	0.5	15.5
	nph3.hmm	16.2	-4.1	-2.3	0.7	16.4
	average	**15.7**	**-0.9**	**-0.6**	**0.6**	**15.9**
bzip2-1.0.6	chicken.jpg	0.8	0.0	0.0	1.0	0.2
	liberty.jpg	0.7	0.0	0.0	0.7	0.5
	input.program	**14.2**	0.5	0.0	10.3	13.9
	text.html	3.5	0.0	0.0	2.1	4.7
	input.source	10.5	0.0	-0.8	7.9	9.7
	input.combined	13.2	0.0	-0.7	9.5	12.5
	average	**7.2**	**0.1**	**-0.3**	**5.2**	**6.9**
Chombo (1 proc)	common.input	6.6	8.2	20.3	2.4	6.0
NWChem (32 procs)	aug-cc-pvdz	67.2	-	-	13.2	63.5

Table 5.3 : Performance statistics for various codes after eliminating dead writes.

from Fortran to C code for our experiments. This benchmark has very long basic blocks that stress the register usage. The `fpppp` code stresses the code segments we modified. Despite being a pathological case, our modified `gcc` showed a 2% speedup when compiling `fpppp`.

5.7.2 Case study: 456.hmmer

`456.hmmer` benchmark is a computationally intensive program. It uses profile hidden markov models of multiple sequence alignments, which are used in computational biology to search for patterns in DNA sequences.

It was surprising to see that `456.hmmer` had only 0.3% deadness in the unoptimized case, whereas it had 30% deadness in the optimized case for the `GNU` compiler (see Table 5.1). In fact, the absolute number of dead writes increased by 54 times from the unoptimized to the highest optimized code for the `nph3.hmm` workload. `Intel` and `PGI` compilers also showed disproportionate rises in deadness for optimized cases.

Listing 5.3 shows the code snippet where DeadSpy identified high-frequency dead writes for the `-O2` optimized code. This code appears in a two-level nested loop, and DeadSpy reported that the write in `line 3` overwrote the write in `line 1`. In the unoptimized code,

```
1       ic[k] = mpp[k] + tpmi[k];
2       if ((sc = ip[k] + tpii[k]) > ic[k])
3         ic[k] = sc;
```

Listing 5.3: Dead writes in `403.hmmer`.

```
1       int icTmp = mpp[k] + tpmi[k];
2       if ((sc = ip[k] + tpii[k]) > icTmp)
3         ic[k] = sc;
4       else
5         ic[k] = icTmp;
```

Listing 5.4: Avoiding dead writes in `403.hmmer`.

the two writes to `ic[k]` one each on `line 1` and `line 3` are separated by a read in the conditional expression on `line 2`, thus making them non dead. In the optimized code, the value of `ic[k]` computed on `line 1` is held in a register, which is reused during the comparison on `line 2`; however, the write to memory on `line 1` is not eliminated, since the compiler cannot guarantee that the arrays `ip, tpii` and `ic` do not alias each other. Thus in the optimized code, if `line 3` executes, it kills the previous write to `ic[k]` on `line 1`.

On inspecting the surrounding code, we inferred that the three pointers always point to different regions of memory and never alias each other, thus making them valid candidates for declaring as `restrict` pointers in `C` language. However, we found that the `gcc 4.1.2` compiler does not fully respect the `restrict` keyword. Hence, we hand optimized the code as shown in Listing 5.4. The optimization improved the running time by more than 15% on average. Table 5.3 shows the reduction in other resources.

This pattern of deadness repeated several times in the same function. `Intel 11.1` compiler at its default optimization level honors the `restrict` keyword; hence, we used it for comparison. We observed that the enclosing function had 13 non-aliased pointers. Declaring these 13 pointers as `restrict` improved the running time by more than 40% on average (`L1` misses, `L2` misses, instructions executed, and cycle count reduced respectively by 34%, 47%, 45%, and 40%). The Intel compiler performed dramatically better because of efficient SIMD vectorization via SSE instructions once the pointers were guaranteed to not alias each other. On disabling vectorization, we observed 16% speedup; nevertheless, DeadSpy *pinpointed optimization limiting code regions, indicating opportunities for performance improvement.*

```
1  Bool mainGtU ( UInt32  i1, UInt32  i2, UChar*  block, ...) {
2    Int32  k; UChar  c1, c2; UInt16 s1, s2;
3    /* 1 */
4    c1 = block[i1]; c2 = block[i2];
5    if (c1 != c2) return (c1 > c2);
6    /* 2 */
7    i1++; i2++; c1 = block[i1]; c2 = block[i2];
8    if (c1 != c2) return (c1 > c2);
9    /* 3 */
10   i1++; i2++; c1 = block[i1]; c2 = block[i2];
11   if (c1 != c2) return (c1 > c2);
12   ... 12 such checks ...
13   ... rest of the function ...
14 }
```

Listing 5.5: Dead writes in `bzip2` at `mainGtU()`.

```
1  leal (%r11,%rcx), %r8d  #%r8d contains i1
2  leal 1(%r8), %ebx       #compute (i1+1) in %ebx
3  leal 2(%r8), %r9d       #compute (i1+2) in %r9d
4  movq %rbx, 360(%rsp)    #spill (i1+1) to stack
5  movq %r9, 352(%rsp)     #spill (i1+2) to stack
6  #... (i1+3) to (i1+11) are computed and spilled ...
7  #... first check of if(c1 != c2)
8  cmpb %al, (%r15,%rdx) #if (c1 != c2)
9  ...
10 #... second check of if(c1 != c2)
11 cmpb %al, (%r15,%rdx) #if (c1 != c2)
12 ...
```

Listing 5.6: Hoisting and spilling in `bzip2`.

5.7.3 Case study: bzip2-1.0.6

`bzip2` is a widely used compression tool. For our experiments on `bzip2`, we used the most recent publicly available version `1.0.6` with the same workload files as SPEC CPU2006. We did not use `401.bzip2` from SPEC CPU2006 since inlining is disabled in that version, presumably for code portability reasons. Enabling inlining on `401.bzip2` produces the same issue as discussed here.

DeadSpy reported frequent dead writes in the inlined function `mainGtU()` shown in Listing 5.5. In this function, 12 conditions are successively checked, each of which accesses the array elements `block[i1]`$\cdots$`block[i1+11]` and `block[i2]`$\cdots$`block[i2+11]`. The function returns if any one of the check fails. The corresponding x86 assembly in Listing 5.6 shows that `gcc 4.1.2` hoists the computation of indices `(i1+1)`$\cdots$`(i1+11)` ahead of the first conditional on `line 8`. `Lines 2` and `3` in Listing 5.6 show sample instructions computing `(i1+1)` and `(i1+2)`. On the register starved x86 architecture, the precomputed values could not be kept in registers and hence they are all spilled. `Lines 4` and `5` in Listing 5.6 show sample spill code for `(i1+1)` and `(i1+2)`. The compute and spill pattern repeats unconditionally

```
1  /* value unknown at compile time */
2 extern int gDisableAggressiveSched;
3 Bool mainGtU (...) {
4    ...
5    switch(gDisableAggressiveSched){
6    case 1: // always taken
7      c1 = block[i1]; c2 = block[i2];
8      if (c1 != c2) return (c1 > c2);
9    case 2: // Fall through
10     i1++; i2++; c1 = block[i1]; c2 = block[i2];
11     if (c1 != c2) return (c1 > c2);
12   case 3: // Fall through
13     i1++; i2++; c1 = block[i1]; c2 = block[i2];
14     if (c1 != c2) return (c1 > c2);
15   ... 12 such checks ...
16   } // end switch
17   ... rest of the function ...
18 }
```

Listing 5.7: `bzip2` modified to eliminate dead writes arising from hoisting and spilling.

12 times before the first conditional test. During the execution of the `mainGtU()` function, based on the data present in `block`, often the code fails one of the early conditional tests. In this case, all the pre-computed values are unused; and the spilled values are an additional overhead. DeadSpy detects that these spill slots are written repeatedly and unread. Since `mainGtU()` happens to be at the heart of `bzip2`'s compute kernel, the effect of hoisting index computations into the hot code region has a negative impact on performance. This behavior exposes *the lack of cooperation between the instruction scheduling and register allocation phases of* `gcc`, which results in poor generated code.

To suppress `gcc`'s over-aggressive scheduling and register spilling, we overlaid a switch statement over the control flow—a technique analogous to Duff's device [68]. Listing 5.7 shows restructured code where the value of `gDisableAggressiveSched` is always `1` at run-time making the `case 1` arm of the `switch` statement to be the always taken branch.

Table 5.3 shows the improvements obtained using our workaround. `bzip2` shows an average speedup of 7.2% with the maximum of 14.2% for the `input.program` workload. While the cache misses remained almost the same before and after the fix, the operations performed and the cycle counts reduced by a proportion commensurate with the observed performance gains. This statistics is justifiable since the code changes eliminated dead writes to on-stack temporaries, which are typically cached. Avoiding the unnecessary index computation and spilling reduced the execution time.

```
1  Wgdnv(i,j,k,0) = a1
2  Wgdnv(i,j,k,inorm) = b1
3  Wgdnv(i,j,k,4) = c1
4  if (spout.le.(0.0d0)) then
5    Wgdnv(i,j,k,0) = a2
6    Wgdnv(i,j,k,inorm) = b2
7    Wgdnv(i,j,k,4) = c2
8  endif
9  if (spin.gt.(0.0d0)) then
10   Wgdnv(i,j,k,0) = a3
11   Wgdnv(i,j,k,inorm) = b3
12   Wgdnv(i,j,k,4) = c3
13 endif
```

Listing 5.8: Dead writes in a `Chombo` Riemann solver.

```
1  if (spin.gt.(0.0d0)) then
2    Wgdnv(i,j,k,0) = a3
3    Wgdnv(i,j,k,inorm) = b3
4    Wgdnv(i,j,k,4) = c3
5  else if (spout.le.(0.0d0)) then
6    Wgdnv(i,j,k,0) = a2
7    Wgdnv(i,j,k,inorm) = b2
8    Wgdnv(i,j,k,4) = c2
9  else
10   Wgdnv(i,j,k,0) = a1
11   Wgdnv(i,j,k,inorm) = b1
12   Wgdnv(i,j,k,4) = c1
13 endif
```

Listing 5.9: Avoiding dead writes in Riemann solver.

5.7.4 Case study: Chombo's amrGodunov3d

We ran DeadSpy on `amrGodunov3d`, a standard benchmark program that uses `Chombo` [11], which is a framework for solving partial differential equations in parallel using block-structured, adaptively refined grids. For detecting node-level inefficiency, we ran `amrGodunov3d` on a single node[2] and detected 34% deadness.

The Chombo framework lacks *design for performance*; it is a hybrid code which has a `C++` driver with computational kernels written in `Fortran`. We discuss two frequent dead write scenarios, where the developer did not pay enough attention to performance while designing the framework.

First, the code snippet shown in Listing 5.8 appears in a 3-level nested loop of the computationally intensive `Riemann solver` kernel, which works on a 4D array of 8-byte `real` numbers. DeadSpy reported that the writes in `lines 1, 2,` and `3` were killed by the writes in `lines 5, 6` and `7`, respectively. In addition, the same writes in `lines 1, 2,` and `3` were killed by the writes in `lines 10, 11` and `12`, respectively. Furthermore, the writes

[2]We used Intel Xeon E5530 processor and Intel compiler tools version 12.0.0.

```
1  // Temporary primitive variables
2  FArrayBox WTempMinus(WMinus[dir1].box(),...);
3  FArrayBox WTempPlus(WPlus [dir1].box(),...);
4  FArrayBox AdWdx(WPlus[dir1].box(),...);
5  // Copy data for in place modification
6  WTempMinus.copy(WMinus[dir1]);
7  WTempPlus.copy(WPlus [dir1]);
8  m_gdnvPhysics->quasilinearUpdate(AdWdx,...);
```

Listing 5.10: `Chombo` dead writes in C++ constructors.

in `lines 5, 6` and `7` were killed by the writes in `lines 10, 11` and `12`, respectively. To fix the problem, we applied a trivial code restructuring by using `else if` nesting as shown in Listing 5.9.

Second, the code snippet shown in Listing 5.10 appears on a hot path with 2-level nested loop. The call to construct `FArrayBox` objects — `WTempMinus` and `WTempPlus`, on `lines 2, 3` respectively, zero initialize a 4D-box data structure member in `FArrayBox`. The calls to the `copy()` member function on `lines 6,7` fully overwrite the previously initialized 4D-box data structures with new values. Similarly, 4D-box `AdWdx` constructed and zero initialized in `line 4` is fully overwritten inside the function `quasilinearUpdate()` called from `line 8`. These dead writes together contribute to a deadness of 20% in the program. The dead writes result from *using a non-specialized constructor*. We remedied the problem by overloading the constructor with a specialized leaner version which did not initialize the 4D box inside `FArrayBox`.

In `Chombo`, the patten of dead writes caused by initializations followed by overwrites was pervasive. In fact, we observed this pattern in SPEC CPU2006 benchmarks also, we omit the details for brevity. By using slimmer versions of constructors in five more similar contexts, we improved `Chombo`'s running time by 6.6%. Table 5.3 shows reductions in cache misses and total instructions as well.

5.7.5 Case study: NWChem's aug-cc-pvdz

In this subsection, we study the impact of dead writes on the scalability and performance of NWChem [230]—a computational chemistry code widely used in the scientific community. NWChem provides a portable implementation of several Quantum Mechanics and Molecular Mechanics methods: Coupled-cluster (QM-CC), Hartree-Fock (HF), Density functional the-

```
237 c getting piece of atomic 2-e integrals (mu nu | lambda sigma)
238 c zeroing ---
239       call dfill(work1, 0.0d0, dbl_mb(k_work1), 1)
240       call dfill(work2, 0.0d0, dbl_mb(k_work2), 1)
```

Listing 5.11: Largest contribution to dead writes in a `NWChem` in `tce_mo2e_trans_` routine.

ory (DFT), Ab initio molecular dynamics (AIMD) etc. We executed high-accuracy QM-CC, where many of the NWChem computational cycles are spent in scientific runs. We conducted our experiments on a 2-socket 8-core 2-way SMT Intel Xeon E5-2650 CPU clocked at 2.0 GHz attached to a 48 GB DDR3 memory. We used the QM-CC `aug-cc-pvdz` input for our studies. This is a relatively smaller input that is representative of actual scientific executions of NWChem. We compiled NWChem to use MPI using MPICH version 3.04 [15]. We launched NWChem by spreading MPI processes maximally on the node to reduce memory congestion.

DeadSpy quantified about 50% memory writes in this configuration of NWChem as dead. 88% of the dead writes (44% of total writes) were happening in the function `dfill_`, that zero initializes two arrays—`work1` and `work2`. Most of the dead writes were in initializing the `work2` array. Most of the dead and the killing writes happened in the same location. The source snippet for the function `tce_mo2e_trans_`, which calls `dfill_` is shown in Listing 5.11. A pair of call chains leading to the most frequent dead and killing writes is shown in Figure 5.7. Fewer dead writes happened to small parts of the same buffer at other places, one of which is shown in Figure 5.8.[3] DeadSpy identified a total of 1.5TB of dead writes to the buffer, aggregated across all processes during the entire execution.

Table 5.4 shows the strong scaling execution time of NWChem for 2, 4, 8, 16, and 32 processes. Row #2 shows the running time for the original program that had dead writes. Row #3 shows the running time for the optimized program after the dead writes are removed. Figure 5.9 plots the performance of NWChem for 2, 4, 8, 16, and 32 processes with dead writes and after eliminating the dead writes.

Table 5.5 shows the last-level (L3) cache misses in the original program with dead writes. Row #2 shows the cache misses for the entire program. Row #3 shows the rise in last level

[3]Note that the call chains shown were obtained after we enhanced DeadSpy with call-site level attribution discussed in Chapter 6.

```
 ---------------------------------------------------------------------------
dfill_:src/util/dfill.f:12
 tce_mo2e_trans_:src/tce/tce_mo2e_trans.F:240
  tce_energy_:src/tce/tce_energy.F:1326
   tce_energy_fragment_:src/tce/tce_energy_fragment.F:88
    task_energy_doit_:src/task/task_energy.F:275
     task_energy_:src/task/task_energy.F:111
      task_:src/task/task.F:356
       MAIN__:src/nwchem.F:263
 *********************************Killed By*********************************
dfill_:src/util/dfill.f:12
 tce_mo2e_trans_:src/tce/tce_mo2e_trans.F:240
  tce_energy_:src/tce/tce_energy.F:1326
   tce_energy_fragment_:src/tce/tce_energy_fragment.F:88
    task_energy_doit_:src/task/task_energy.F:275
     task_energy_:src/task/task_energy.F:111
      task_:src/task/task.F:356
       MAIN__:src/nwchem.F:263
 ---------------------------------------------------------------------------
```

Figure 5.7 : Bottom-up view of the call-paths leading to the most frequent dead and killing writes in NWChem.

```
 ---------------------------------------------------------------------------
dfill_:src/util/dfill.f:12
 tce_mo2e_trans_:src/tce/tce_mo2e_trans.F:240
  tce_energy_:src/tce/tce_energy.F:1326
   tce_energy_fragment_:src/tce/tce_energy_fragment.F:88
    task_energy_doit_:src/task/task_energy.F:275
     task_energy_:src/task/task_energy.F:111
      task_:src/task/task.F:356
       MAIN__:src/nwchem.F:263
 *********************************Killed By*********************************
hf2mkr_:src/NWints/int/hf2mkr.F:155
 hf2_:src/NWints/int/hf2.F:324
  int_2e4c_:src/NWints/api/int_2e4c.F:358
   tce_mo2e_trans_:src/tce/tce_mo2e_trans.F:243
    tce_energy_:src/tce/tce_energy.F:1326
     tce_energy_fragment_:src/tce/tce_energy_fragment.F:88
      task_energy_doit_:src/task/task_energy.F:275
       task_energy_:src/task/task_energy.F:111
        task_:src/task/task.F:356
         MAIN__:src/nwchem.F:263
 ---------------------------------------------------------------------------
```

Figure 5.8 : Bottom-up view of the call-paths leading to a less frequent dead and killing writes in NWChem.

No. Processes	2	4	8	16	32
Execution time with dead writes (seconds)	236	139	97.9	83.4	97.3
Execution time without dead writes (seconds)	200	120	78.1	50.4	58.2
Speed-up	1.18×	1.16×	1.25×	1.66×	1.67×

Table 5.4 : Impact of dead writes on NWChem's execution time.

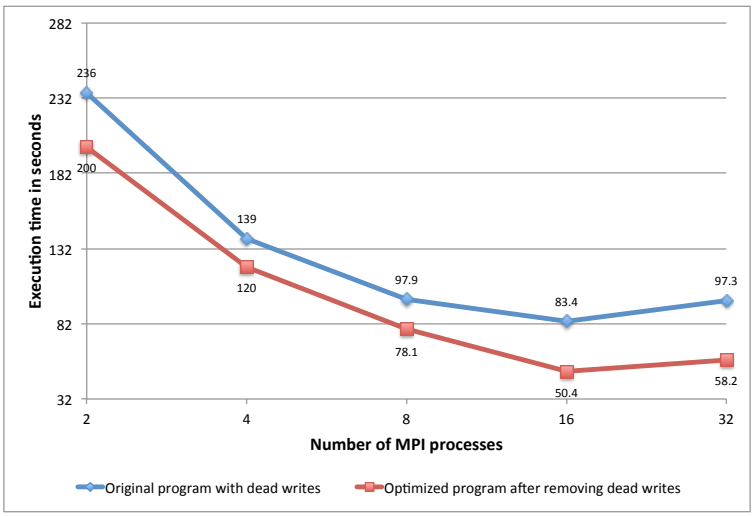


Figure 5.9 : NWChem performance before and after removing dead writes.

No. Processes	2	4	8	16	32
Program-wide L3 cache misses	1.42E+08	5.52E+08	1.73E+09	4.69E+09	6.09E+09
Rise in L3 cache misses wrt 2 processes	1×	3.89×	12.2×	33.0×	42.9×
L3 cache misses in `dfill_` (% total L3 misses)	1.68E+07 (12%)	3.07E+07 (5.6%)	1.00E+09 (58%)	3.53E+09 (75%)	4.09E+09 (67%)
Rise in L3 cache misses in `dfill_` wrt 2 processes	1×	1.83×	59.5×	210×	244×

Table 5.5 : Last-level cache misses in NWChem with dead writes

cache misses for the entire program with respect to the cache misses in the program with only 2 processes. Row #4 shows the cache misses for the `dfill_` routine in the context where maximum dead writes occurred, along with the percentage contribution to the L3 cache misses in the entire program. Row #5 shows the rise in last level cache misses for the `dfill_` routine with respect to the cache misses in the `dfill_` routine with only 2 processes. Clearly, the L3 cache misses grow as the number of processes increases, leading to catastrophic performance loss. Most of the L3 cache misses come from the `dfill_` routine where the dead writes occurred. The rise in L3 misses in the `dfill_` routine is meteoric with the rise in the number of processes. This observation shows that the dead writes caused by the `dfill_` routine are a major source of performance loss.

The performance loss increases with increasing processor count. The performance loss is significant beyond 8 processors; this is because, each Intel Xeon E5-2650 has 4 memory

channels (total of 8 in a 2-socket system) and once the number of processes crosses 8, the memory bandwidth saturates since all cores simultaneously try to zero initialize a large buffer. This phenomenon can be indirectly inferred via the observed rise in L3 cache misses.

By consulting NWChem experts, we identified that the buffer size was larger than necessary and the zero initialization was unnecessary, which were causing the dead writes. Subsequently, NWChem developers eliminated the unnecessary initialization from the code base. Eliminating the dead writes speed-up NWChem by up to 1.67× for the `aug-cc-pvdz` input.

5.8 Discussion

In this chapter, we demonstrated the value of identifying dead writes in an execution. Dead writes are a common symptom of inefficiency. Performance losses due to dead writes arise from developers' inattention to performance, poor choice of algorithms and data structures, and ineffective compiler optimizations, among others.

Compiler developers need to choose compiler internal data structures and algorithms prudently. Compiler developers need better profitability analysis to understand when optimizations may do more harm than good.

Profiling dead writes is only one form of many other possibilities of fine-grained execution profiling. Profiling for wasteful resource consumption opens a new avenue for application tuning. In follow-on work, we successfully identified additional performance tuning opportunities by using fine-grained profiling of redundant writes (writing an unmodified value to the same location) and redundant computations [236].

Finally, our experience with DeadSpy highlighted the need for associating calling context to execution monitoring. Attributing metrics to their execution contexts is important in providing insightful feedback in fine-grained monitoring tools. In the next chapter, we improve upon the contextual attribution strategy used by DeadSpy to create a general framework for fine-grained contextual attribution for use by other monitoring tools.

Chapter 6

Attributing Fine-grain Execution Characteristics to Call-Paths

Knowledge of the self is the mother of all knowledge. So it is incumbent on me to know my self, to know it completely, to know its minutiae, its characteristics, its subtleties, and its very atoms.

Kahlil Gibran

Large software systems leverage application frameworks, multiple layers of libraries, or both. A library is often used in multiple different contexts by a program. It is essential, hence, to attribute execution characteristics in full calling context to distinguish the behavior of a code from one context to another. Context is particularly important in the presence of language features such as C++ templates that invoke the same source code with different instantiations. A recurring theme in this dissertation is attributing the runtime execution characteristics to *source code* and *data* in context.

Contextual attribution of the causes of idleness, discussed in Chapter 2, enhanced the call-path collection in HPCTOOLKIT [2] for GPU tasks. Contextual attribution of redundant barriers, discussed in Chapter 3, relied largely an on off-the-shelf unwinder such as `libunwind` [161]. Both these approaches collect call-paths relatively infrequently. Our work on fine-grained execution monitoring in the context of the DEADSPY tool, discussed in Chapter 5, highlights the need for collecting calling contexts at a very high frequency, e.g., every memory access instruction. We claim that high-frequency call-path collection is a necessary component in various other fine-grain execution monitoring tools. In this chapter, we build a library module that provides a space and time efficient implementation of ubiquitous source and data context collection needed by fine-grain execution profiling tools.

6.1 Motivation and overview

Tracking program execution at every machine instruction is a popular technique for identifying several classes of software issues. Two of the most common applications of such fine-grain execution monitoring (FEM) are *execution characterization* and *correctness checking*. For execution characterization, FEM is employed for reuse-distance analysis [135], cache simulation [103], computational and memory redundancy detection [42], as well as power and energy analysis [131, 238], to name just a few applications. For correctness checking, FEM is employed for taint analysis [31], concurrency bug detection [80, 130, 141], and malware analysis [162], among other applications. FEM is also employed in hardware simulation [36], reverse engineering [132], software resiliency, testing, tracing, debugging [235], and execution replay [166, 186].

Two approaches for FEM are *static* and *dynamic* binary instrumentation. Several frameworks [32, 144, 168] support static and/or dynamic binary rewriting. By their very nature, techniques for FEM add non-trivial overhead. FEM overhead is often higher using dynamic rewriting. Tool-specific code invoked for instrumented instruction(s) is often called an *analysis routine*. Since each instruction in the original program can potentially invoke an analysis routine during monitoring, tool writers strive to limit the overhead of their analysis routines. Naturally, performing less work per analysis routine lowers a tool's runtime overhead. The less measurement data a tool gathers, however, the less insightful its analysis and feedback are likely to be.

Intel's Pin [144] is a leading dynamic binary instrumentation framework for developing FEM tools. *A key feature missing in Pin is the ability to provide the call-path at any point inside an analysis routine.* Naively, an FEM tool can log each instruction executed to a trace file and reconstruct full call-paths with post-mortem analysis. This approach, however, would generate huge logs with high runtime overhead, making it infeasible to use on executions of any significant length. Profiling to collect aggregate metrics is preferred over tracing for long running executions. Determining the calling context using stack unwinding on each instruction introduces excessive overhead and is unsuitable for FEM tools. Maintaining a shadow stack throughout the execution is a superior approach since it accelerates the common

case—frequent call-path collection. There have been a few earlier efforts to collect call-path information at instruction-level granularity; however, they suffer from various limitations such as call-path inaccuracy and runtime overhead, making them less useful in practice.

To support fine-grained attribution of metrics to calling contexts and data objects, we developed CCTLib. CCTLib is a call-path collection library for Pin that employs a shadow stack to support on-demand use of call-paths instead of logging or call stack unwinding. CCTLib maintains information about call-paths efficiently and compactly in a *calling context tree* (CCT) [9]. CCTLib is modest in overhead and usable in practice for reasonably long-running programs. CCTLib collects accurate call-paths even through dynamically loaded libraries, stripped libraries, and executable code for which the compiler recorded incorrect or incomplete information about function bounds. CCTLib's operation is largely transparent; it requires only initialization at the start-up and calling an interface procedure at any point in a Pin analysis routine to inspect the call-path. CCTLib can be used by any Pin-based FEM tool. CCTLib is for FEM tools only, and it should not be confused with call stack tracking or unwinding capabilities used by performance tools that attribute execution costs using timers and hardware performance counters.

Attributing metrics to source in context is only one facet of program monitoring. While attributing metrics to source in context identifies problematic code, the same code and context might be accessing different data objects. Different data objects can make the same code exhibit different performance characteristics [133, 134, 135, 136, 138]. Fine-grained data-centric attribution is a technique that associates memory location(s) involved in each instruction with their corresponding data objects (e.g., static, stack, and heap variables). Pinpointing problematic data objects is an important aspect of execution analysis. Moreover, data-centric information is necessary for optimizing memory reuse distance and eliminating false sharing in multi-threaded programs. CCTLib supports both context- and data-centric attributions. CCTLib's data-centric attribution leverages the calling context information as well. Figure 6.1 provides a schematic view of CCTLib.

With CCTLib, we demonstrate that collecting call-paths on each executed instruction is possible, even for reasonably long-running programs. Compared to other open-source Pin tools for call-path collection, CCTLib provides richer information that is accurate even for

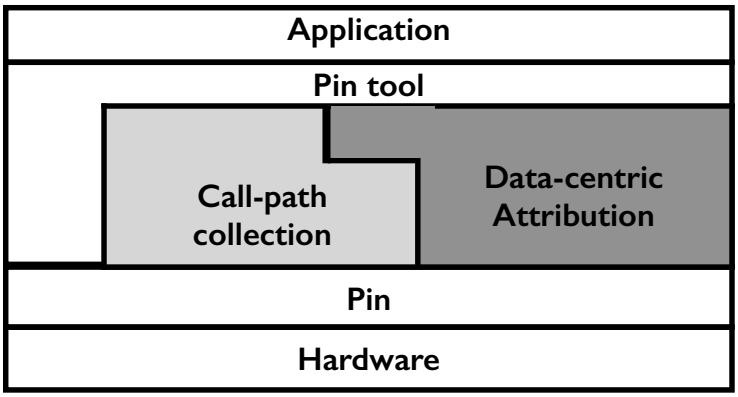


Figure 6.1 : CCTLib schematic diagram. A fine-grained analysis tool built on Pin can transparently use CCTLib to attribute metrics to call-paths and data objects. CCTLib is composed of the call-path collection component and the data-centric attribution component. The data-centric attribution component uses the call-path collection component.

programs with complex control flow and does so with about 30% less overhead—a difference of 14× on average. CCTLib enables the attribution of metrics in Pin tools to both code and data.

6.2 Contributions

This work, which appeared in the Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2014 [41], makes the following contributions.

1. It describes the design and implementation of CCTLib—an open-source framework for accurate and efficient collection and attribution of context- and data- centric information in Pin tools,

2. It enables more accurate and informative FEM tools than prior work,

3. It improves upon our prior call-path collection technique described in chapter 5 by about 30%, and

4. It demonstrates the utility of CCTLib's context- and data-centric capabilities by collecting such information on each executed instruction for a suite of long-running programs.

6.3 Chapter roadmap

The rest of the chapter is organized as follows. Section 6.4 provides the necessary background for our work. Section 6.5 describes our methodology for call-path collection. Section 6.6 describes the implementation of CCTLib. Section 6.7 evaluates our approach. We end this chapter with a discussion in Section 6.8.

6.4 Background

In this section, we discuss the state-of-the-art in call-path collection, data- centric attribution, and outline challenges maintaining calling context information in Pin.

6.4.1 Call-path collection techniques

There are two principal techniques used to collect call-paths in an execution: unwinding the call stack and maintaining a shadow call stack.

Call stack unwinding, on `x86` architectures, walks procedure frames on the call stack using either compiler-recorded information (e.g., libunwind [161]) or results from binary analysis (e.g., HPCTOOLKIT [2]). Stack unwinding does not require instrumentation and it does not maintain state information at each call and return. As a result, it adds no execution overhead, except when a stack trace is requested. This technique is well suited for coarse-grained execution monitoring, e.g., sampling-based performance tools and debuggers. However, applying call stack unwinding to gather calling context information for each machine instruction executed would frequently gather slowly changing calling context at unacceptably high overhead.

Stack shadowing involves maintaining calling context as the execution unfolds; this can be accomplished by either instrumenting every function entry and exit either at compile time or using binary rewriting. Stack shadowing is used by tools including Scalasca [75] and TAU [147]. The advantage of stack shadowing is that at every instant, the stack trace is ready, and hence it can be collected in a constant time. Stack shadowing is well suited for FEM tools. A disadvantage of stack shadowing is that instrumentation adds overhead. Furthermore, it is not stateless, and it requires extra space to maintain the shadow stack.

6.4.2 Pin and call-path collection

Intel's Pin [144], a leading binary rewriting framework for FEM tools, does not provide any API for collecting call-paths. The work presented in this chapter overcomes this limitation of Pin. Tools that use Pin are called *Pin tools*.

Providing access to an application's call-path, while executing analysis routines in a Pin tool, poses several challenges. First, unwind libraries can't be employed out of the box since Pin Just-In-Time (JIT) compiles code from the original binary and executes the JIT-compiled code from its code cache. Second, employing call-path unwinding for FEM is prohibitively expensive. Finally, even if unwinding were possible in the JITed code, transitions between the Pin framework and application code would clutter the call stack with unwanted information.

6.5 CCTLib methodology

CCTLib primarily employs stack shadowing for collecting call-paths. We recognize two key aspects in collecting call-paths: accuracy and efficiency.

6.5.1 Call-path accuracy

Stack shadowing is typically performed by inserting instrumentation at function entries and exits. For binary instrumentation, function entries are typically discovered via symbol information and function exits are discovered via static instruction disassembly. Instrumenting function entries and exits can be inaccurate when function symbols are missing or wrong [224] and when machine code disassembly is incorrect because of incorrect assumptions about instruction boundaries. Despite significant efforts to perform accurate static `x86` disassembly [83], the technique is not foolproof.

Due to the stateful nature of stack shadowing, instrumentation of function entries and exits must match accurately. A single mismatched entry or exit can corrupt the shadow stack leading to incorrect call-paths for rest of the execution. We conducted an experiment using Pin to identify the potential impact of imprecision in binary analysis on call-path collection. Our experiment did the following:

1. It computed the fraction of instructions executed at runtime that did not fall under

Application	Instructions not detected as part of any range(col2)	Stack-size affecting instructions belonging to a range but missed due to incorrect disassembly (col3)	Total error in call-path collection (col2 + col3)
LULESH-OpenMP[1]	93%	5.6e-02%	93%
LULESH-CUDA[2]	17.3%	2.04e-01%	17.5%
LLVM-OPT compiling bzip2[2]	4.17e-07%	8.28%	8.28%
tar (Linux command-line utility)	22.0%	4.1e-1%	22.3%
ls (Linux command-line utility)	11.7%	5.0e-1%	12.2%
Xalan (SPEC CPU2006 [85])[2]	2.78e-05	4.26%	4.27%
omnetpp(SPEC CPU2006 [85])[2]	7.08e-04%	3.49%	3.49%

[1] Compiled with `Intel icpc v13.0.0, -O2`. [2] Compiled with `gcc 4.1.2, -Os`.

Table 6.1 : Impact of incorrect/incomplete static disassembly on call-path collection in Pin.

any function range discovered via Pin's static disassembly, and

2. It computed the fraction of stack-size-affecting instructions that are executed at runtime for which the disassembly was incorrect (i.e., start address of the instruction fell in the middle of another instruction during static disassembly).

Table 6.1 shows some applications with a non-trivial amount of imprecision in disassembly.[1] We did not intentionally strip any symbol from an executable. However, we are aware that Nvidia's CUDA libraries used by LULESH-CUDA [112] have symbols stripped. If we stripped executables, incorrect disassembly would be more frequent. For example, for a stripped version of the SPEC CPU2006 `omnetpp` reference benchmark [85], the incorrect disassembly was about 60%. Working with stripped applications (just not stripped libraries) is a valuable use case for software security. High disassembly errors for LULESH-OpenMP [112] are likely due to the following reason: in optimized code generated by recent Intel's `icpc` compiler version 13.0.0, machine code for outlined functions associated with OpenMP parallel regions and loops appears in the executable amidst machine code for the function in which the regions or loops originated. As a result, the machine code for the enclosing function is split into discontinuous pieces by the embedded outlined code. Furthermore, code for embedded outlined functions is not labeled with function symbols.

An alternate method for building shadow stack uses instrumentation of `call` and `return` instructions instead of function entries and exits. If we track every instruction, we can't possibly miss any `call` or `return` instruction, and this ensures accurate shadow stacks. One

[1]The tests were conducted on a variety of modern 64-bit `x86` Linux machines.

can instrument and monitor every instruction in Pin via *trace instrumentation*. A *trace*, in Pin jargon, is a single entry multiple exit code sequence—for example, a branch starts a new trace at the target, and a `call`, `return`, or `jump` ends the trace. Trace instrumentation happens immediately before a code sequence is first executed. Trace instrumentation has the advantage of not missing the instrumentation of any executed instruction. In the rest of the paper, we use the term *Pin-trace* to refer to Pin's traces to distinguish it from our internal representation of traces in CCTLib. By leveraging Pin traces, CCTLib avoids relying on the compiler-based information for function boundaries or for machine instruction disassembly. This design eliminates the possibility of missing any `call` and `return`. Consequently, CCTLib's call-path collection is resilient to symbol stripping. CCTLib uses the compiler-based information only to map machine instructions back to source lines. CCTLib relies on symbol information for a limited set of functions such as `pthread_create`; however, such symbols are available in any dynamically linked application. Data-centric attribution with CCTLib, however, relies on symbol information to attribute memory addresses back to static variables.

6.5.2 Call-path efficiency

Since our goal is to support call-path collection on every monitored instruction in an FEM tool, our techniques need to be efficient in both space and time. CCTLib employs CCTs to store call-path information. Common prefixes are shared across all call-paths, which reduces space overhead dramatically. CCTLib pays particular attention to use constant-time operations within its analysis routines to keep overhead low. By employing efficient data structures, e.g., splay trees [215] for maps, our implementation achieves acceptable overhead.

6.6 Design and implementation

In this section, we describe the implementation of CCTLib. Section 6.6.1 describes how we collect a CCT for context-centric attribution. Section 6.6.2 describes the support for data-centric attribution within CCTLib. Table 6.2 lists the key APIs that CCTLib exposes to its

No.	Signature	Description
1	`typedef bool (*TrackInsCallback)(INS ins)`	Client Pin tool callback to determine if the given instruction is to be instrumented.
2	`typedef void (*ClientInstrumentationCallback) (INS ins, VOID *v, OpaqueHandle_t handle)`	Callback to allow the client Pin tools to add their own instruction-level instrumentation.
3	`bool Initialize(TrackInsCallback cb=TRACK_ALL_INS, ClientInstrumentationCallback clientInsCB=0, void * clientCallbackArg=0)`	CCTLib initialization.
4	`ContextHandle_t GetContextHandle(OpaqueHandle_t handle=0)`	Returns a handle that represents the calling context of the current thread.
5	`DatatHandle_t GetDataObjectHandle(void * address)`	Returns a handle that represents the data object accessed by `address`.

Table 6.2 : APIs and data structures exposed by CCTLib.

client Pin tools. `GetContextHandle` obtains the current call-path. `GetDataObjectHandle` obtains the data object accessed at an address.

6.6.1 Collecting a CCT in Pin

CCTLib builds its shadow stack by instrumenting each `call` and `return` machine instruction and stores each call-path in a CCT. The CCT at a given instant has all call-paths seen in the execution up to that point. The path implied by the current node in the CCT to its root represents the calling context at that point. We identify each call-path with a unique 32-bit handle (`ContextHandle_t`) that enables us to reconstruct any call-path during or after program execution.

A particularly challenging part of call-path collection is source-level attribution. At each instruction, Pin provides its instruction pointer (IP), which can be mapped back to source line(s); hypothetically, one can look for this IP in the already recorded IPs under the current node of the CCT. However, such lookup on each instruction is prohibitively expensive. Variable length `x86` instructions, unknown or incorrect function bounds, and tail calls compound the problem. Trace instrumentation, along with a shadow mapping from Pin-traces to their constituent instructions, enables us to solve this problem with a constant-time algorithm.

We describe the necessary instrumentation, key internal data structures, and basic runtime actions performed by CCTLib's analysis routines in Sections 6.6.1.1–6.6.1.3. In Section 6.6.1.4, we contrast CCTLib's internals with DeadSpy's internals. Sections 6.6.1.5–6.6.1.8 describe the details of handling complex control flows. Finally, Section 6.6.1.9 de-

scribes CCTLib's current limitations.

6.6.1.1 Instrumentation

We intercept Pin-trace creation and instrument the following places:

- entry to a Pin-trace.
- each `call` and `return` instruction in the trace, and
- any other instruction(s) the client Pin tool decides to track.

On each Pin-trace creation, we assign a unique identifier (`traceId`) to the trace. Further, we memorize the association between the `traceId` and all instruction addresses in that trace in a map (`ShadowTraceMap`). However, we need not maintain all instructions in the trace in the `ShadowTraceMap`; instead, our map contains only those instructions that are designated as "to be tracked" by the client Pin tool (Table 6.2, row 1) and all `call` and `return` instructions in the trace. The `traceId` is made known to the analysis routine added to each trace entry. Finally, we add instrumentation before each `call` and `return` instruction in the trace. We also instrument `setjmp`, `longjmp`, `pthread_create`, and `_Unwind_SetIP` functions; details about this are described in sections that follow.

6.6.1.2 Supporting data structures

CCTLib's CCT is composed of `TraceNode`s as shown in Figure 6.2. A CCTLib `TraceNode` logically represents a Pin-trace. There is a many-to-one relationship between `TraceNode`s and Pin-traces since the same Pin-trace can be executed from multiple calling contexts. Each `TraceNode` has three fields — 1) an array of `IPNode`s, which we refer to as `IPNodeVec`, 2) `traceId`, and 3) a parent pointer to an `IPNode`. Each element of `IPNodeVec` logically represents an instruction in the corresponding Pin-trace. By default, the size of `IPNodeVec` equals the number of instructions in the corresponding Pin-trace. This mapping enables us to associate every instruction with full calling context whenever the CCTLib client desires; supporting calling context for each instruction forms the highest overhead case. A client Pin tool may tailor CCTLib's tracking of calling contexts for a particular task. For example,

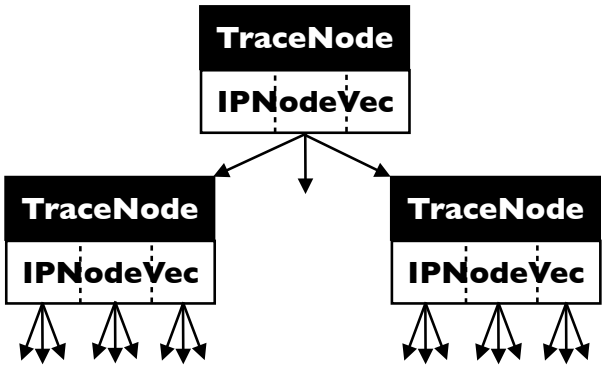


Figure 6.2 : A CCTLib Calling Context Tree.

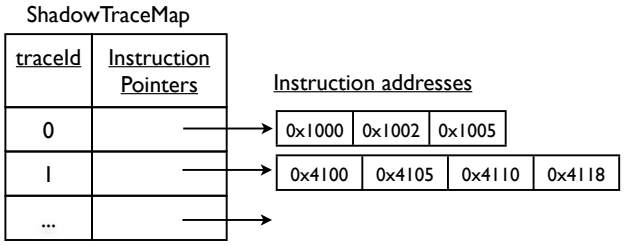


Figure 6.3 : `ShadowTraceMap`: CCTLib's mapping from traces to constituent instructions.

a data race detection tool, which needs to track only memory access instructions, does not require call-paths for non-memory related instructions; in such cases CCTLib can be initialized to track only a client-specified "class" of instructions. CCTLib provides a callback that enables its client Pin tools to specify what instructions need CCT information at trace instrumentation time. No matter what the client Pin tool specifies, CCTLib will always include all `call` and `return` instructions in its `IPNodeVec`. In the best case, the number of entries in an `IPNodeVec` for a Pin trace equals the number of `call` and `return` instructions the trace; this is the lowest overhead case.

There is 1-1 relationship between the N^{th} slot in `IPNodeVec` in a `TraceNode` with `traceId I` and the instruction recorded in the shadow trace map at `ShadowTraceMap[I][N]`. This association enables us to recover the instruction pointer associated with each instruction represented in a `TraceNode` without having to maintain instruction addresses themselves in each `IPNode`. The association is both shared by different `TraceNode`s within a CCT as well as CCTs of different threads. Figure 6.3 shows the `ShadowTraceMap` data structure.

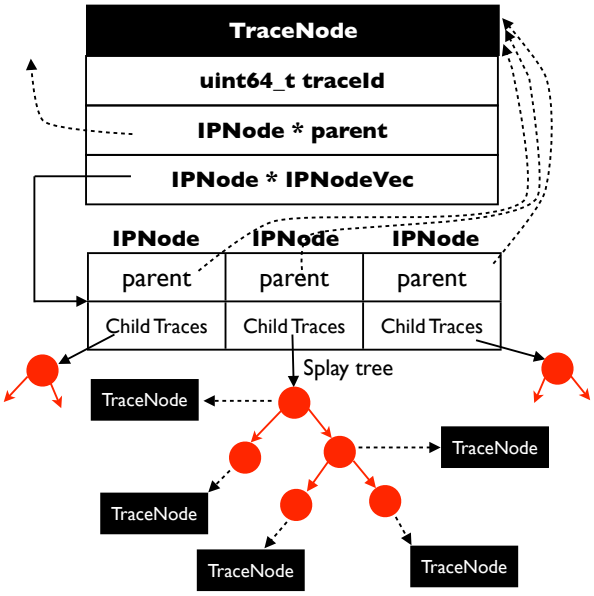


Figure 6.4 : CCTLib's `TraceNode` and `IPNode`.

The details of `TraceNode` and `IPNode` are shown in Figure 6.4. Each element of `IPNode` has two fields. First, a `parent` field which is a pointer to the parent `TraceNode`. Second, a pointer to the root of a splay tree, possibly null. The splay tree represents all traces that executed as callees originating from the given instruction pointer of the given calling context. We allow any instruction to have callees because an exception may happen on any instruction; we represent signal handlers as callees of the instruction where the signal was delivered. Consequently, a data field in every node of the splay tree points to a `TraceNode`. Instructions belonging to a function may be split into several Pin-traces. In our scheme, all Pin-traces belonging to the same function called from the same calling context will appear as different nodes of the splay tree rooted at the same `IPNode`. An intentional consequence of this design choice is that, when an instruction in function B, called from function A performs a *tail call* to function C, the traces in function C will become children traces under function A. Figure 6.5 depicts a tail calling example along with the corresponding CCT.

6.6.1.3 Actions at run time

CCTLib maintains two thread-local variables `curTraceNode` and `curIPNode`. The `curTraceNode` points to the current `TraceNode`. Logically, `curTraceNode` acts as a cursor

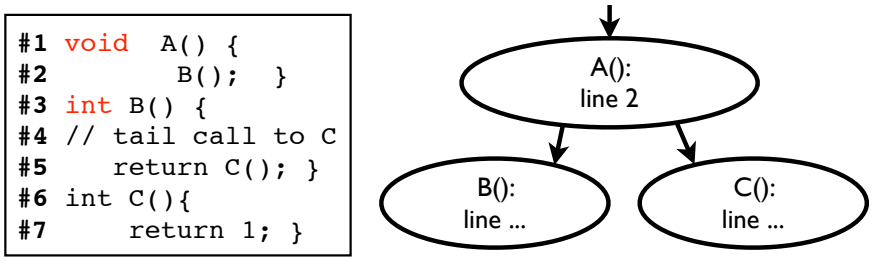


Figure 6.5 : CCTLib CCT for tail calls.

to the currently executing Pin-trace. The `curIPNode` points to a slot in the `curTraceNode`'s `IPNodeVec` that logically represents the current instruction under execution. Logically, `curIPNode` acts as a cursor to the currently executing instruction. If the client Pin tool decides to track only a subset of instructions, `curIPNode` points to the most recently executed instruction in the trace that was chosen for tracking. At runtime, the following five cases arise:

1. Each time a `call` instruction is executed, the analysis routine sets a thread-local flag (`isCall`).

2. When a new Pin-trace with id `traceId` is entered immediately following a `call` instruction (inferred by inspecting the `isCall` flag), it represents a transition from a caller to a callee. In this case, an analysis routine at the trace entry executes the following steps:

   (a) Search for the callee `TraceNode` with key `traceId` in the splay tree rooted at `curIPNode`. If none is found, a new `TraceNode` with `traceId` as the key is created and inserted into the splay tree.

   (b) Set the `curTraceNode` to the callee `TraceNode`.

   (c) Reset the `isCall` flag.

3. When a new Pin-trace with id `traceId` is entered without a `call` instruction being executed, it represents a transition from one trace to the other within the same callee, or possibly a tail call. In this case, the analysis routine at the trace entry executes the following steps:

   (a) Search for the target `TraceNode` with the key `traceId` in the splay tree rooted at `curTraceNode->parent` to find peer traces. If none is found, create and insert into the splay tree a `TraceNode` with `traceId` as key.

   (b) Set `curTraceNode` to the target `TraceNode`.

4. When a `return` instruction executes, set `curTraceNode` to its parent `TraceNode`, simulating the return from a function in the shadow call stack.

5. When any instruction in a trace executes, the analysis routine inserted before that instruction sets `curIPNode` to the corresponding slot of `IPNodeVec`. At trace instrumentation time, the index of the instruction in the Pin-trace is known (which will be its offset in `IPNodeVec`) and hence updating the `curIPNode` to the correct slot within `IPNodeVec` can be done in a constant time even if the execution does not follow a straight line path within a trace.

It is worth mentioning that all cases except 2a and 3a are constant time operations. Our choice of data structures is driven by their amenability to such constant time operations. Cases 2a and 3a are tree lookup operations and can incur a logarithmic complexity. The choice of splay trees ensures that recently accessed `TraceNode`s are near the root of the tree, which makes lookup fast in practice.

As an optimization, the client Pin tool may register a callback that CCTLib calls during trace instrumentation on each instruction designated as "to be tracked" by the client (Table 6.2, row 2). CCTLib passes the index number in `IPNodeVec` that corresponds to the instruction to the client-instrumentation callback (as `opaqueHandle` argument), which the client's analysis routine can pass back to CCTLib's `GetContextHandle()` function when querying the calling context for that instruction. With this technique, CCTLib eliminates the runtime overhead of updating `curIPNode` on each instruction execution; instead, `curIPNode` is derived on demand when the client Pin tool queries the context. The value of `curIPNode` is derived via a constant time indexing operation — `IPNodeVec[opaqueHandle]`; this eliminates the updating of `curIPNode` described in case 5.

At any instruction, its calling context identifier is simply the address of `IPNode` at `curTraceNode->IPNodeVec[opaqueHandle]`. We allocate all `IPNode`s from a fixed memory

pool, hence, instead of pointer-sized (8 bytes on 64-bit machines) handles, we can manage with just 32-bit handles. The handle uniquely represents a call-path in the CCT; by traversing the parents pointers, the entire call-path can be constructed. We allow for a maximum of 2^{32} unique call-paths in a program. CCTLib provides the option to serialize the entire CCT during program termination; the serialized call-path identifiers can be used to extract the corresponding call-paths in a postmortem fashion. CCTLib also provides the option to serialize CCTs as a DOT file for visualization.

6.6.1.4 Contrast with DeadSpy's CCT

Our technique of building CCTs extends our prior work on CCT construction in DeadSpy, discussed in Chapter 5. The CCTs in DeadSpy can only provide source line mapping information for the leaf node of a path; for interior frames DeadSpy can recover only function names. As a consequence, the code in Figure 6.6, which has multiple call sites to the same callee, leads to the ambiguous CCT shown in Figure 6.7. It is unclear from the call-path in Figure 6.7 whether `A` was called from line 2 or line 3 of `main`. CCTLib, in contrast, builds CCT as shown in Figure 6.8. CCTLib has call site level attribution, which disambiguates two calls to `A` from two different call sites in the same caller. The Maid [99] stack trace tool does not have call site level attribution and hence suffers from the same problem.

Another difference between DeadSpy's and CCTLib's CCT construction is the elimination of DeadSpy's notion of `ContextNode` in CCTLib. DeadSpy represents function calls in a CCT using `ContextNode`, which serves as an umbrella for—all callees (`ContextNode`) and traces (`TraceNode`) belonging to a function. On a function call, DeadSpy's scheme employs a two-level search. First, DeadSpy searches the target function under all callees of the current `ContextNode` represented by `childContexts` field, then DeadSpy searches the traces pointed to by `childTraces` in the callee `ContextNode`. DeadSpy uses hash tables as maps for these lookup operations. In contrast, CCTLib consolidates both `ContextNode`s and `TraceNode`s into one `TraceNode`. CCTLib eliminates the two-level lookup with a single-level lookup during a `call` instruction as mentioned before in step 2a. Replacing two-levels of lookup with one-level lookup and employing splay trees, which are better suited for the lookup of frequently accessed items, improves performance as we show in Section 6.7.

```
#1 void  main() {
#2         A(0);
#3         A(1); }
#4 void A(int i) {
#5   StackTraceHere(); }
```

Figure 6.6 : Code that needs line-level disambiguation.

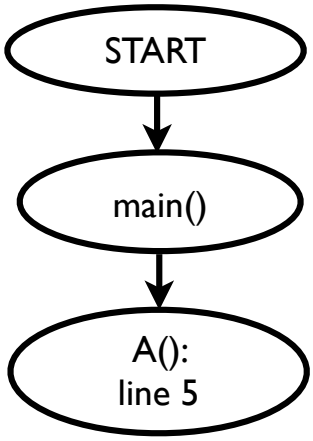


Figure 6.7 : Ambiguous DeadSpy CCT.

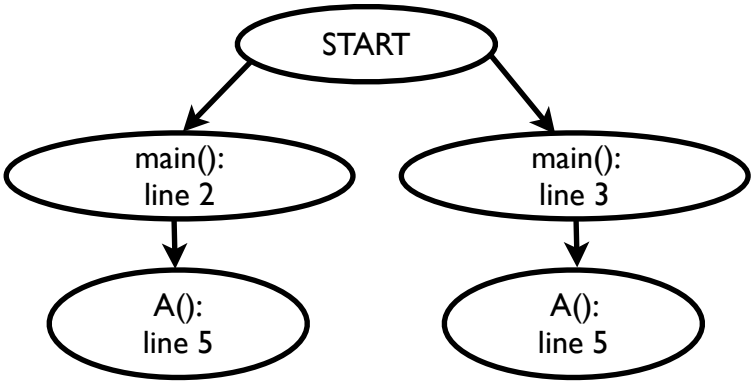


Figure 6.8 : CCT with call site level attribution.

Experiments with the `bzip2` SPEC CPU2006 reference benchmark showed that execution time of monitored programs using splay trees in CCTLib was 50% faster than using google's sparsehash tables [76] and 24% faster than GNU C++'s `std::hash_map`.

6.6.1.5 Signal handling

When a signal is delivered, control asynchronously transfers to a registered signal handler. With CCTLib, the signal handler appears as a callee under the instruction where the signal was delivered. This feature is accomplished by simply setting the `isCall` flag in the analysis routine added using `PIN_AddContextChangeFunction` for signals. All callees of the signal handler will be treated as normal callees. The return from signal handler executes a `return` instruction enabling CCTLib to resume from the position in the CCT where the signal was delivered. If the client tool opts to not track all instructions, the signal handler appears as

if it was called from the last tracked instruction that executed.

6.6.1.6 Handling Setjmp/Longjmp

The `setjmp` and `longjmp` APIs allow programs to bypass the normal call/return flow of control. With `setjmp`, the program memorizes the current architectural state of the program in a user-provided buffer `jmp_buf` and with `longjmp` it restores the state stored in the given buffer. A `longjmp` may unwind multiple frames of the stack without executing a `return`. To ensure CCTLib produces correct call-paths even after `longjmp` calls, CCTLib memorizes the association between the `jmp_buf` and the calling context where `setjmp` was called, and restores the same calling context after a `longjmp`, thus simulating the effect of `longjmp` in its shadow stack.

6.6.1.7 Shadow stack during C++ exceptions

When a C++ program throws an exception, the runtime performs stack unwinding to look for an exception handler. Once a handler is found, execution resumes at the handler, having unwound zero or more stack frames. The C++ unwinder calls `_Unwind_SetIP` as the last step of jumping to the handler function. The `_Unwind_SetIP` has the following signature: `void _Unwind_SetIP(struct _Unwind_Context * context, uint value)`. The first argument (`context`), among other things, contains information about the stack frame that can handle the current exception. It also includes information such as the instruction pointer in the handler's frame that called a chain of functions which eventually led to the exception. The `_Unwind_SetIP` function overwrites its return address with the second argument `value`, which contains the address of the exception handler. Upon return from `_Unwind_SetIP`, control resumes in the exception handler instead of the original caller.

For maintaining a correct shadow stack during an exception, CCTLib instruments the `_Unwind_SetIP` function and uses the information present in the `context` argument to set `curTraceNode` appropriately. On entering `_Unwind_SetIP`, CCTLib captures the first argument, i.e., `context`. CCTLib calls the `_Unwind_GetIP(context)` to obtain the instruction pointer of the call site, `exceptionCallerIP`, in the ancestor frame where the exception handler is present. Now, CCTLib walks up the call chain, starting at `curTraceNode`, looking

for the first `TraceNode` that contains `exceptionCallerIP`. CCTLib stops on the first found `TraceNode` t and records a pointer to it. On the return path from `_Unwind_SetIP`, CCTLib sets its `curTraceNode` to t resulting in the shadow call stack unwinding exactly the same number of frames as the original stack. Subsequent entry to the handler Pin-trace will follow the earlier described strategy under case 3.

This unwinding technique needs tailoring for platforms that employ other strategies for exception handling.

6.6.1.8 CCT in multithreaded codes

CCTLib produces independent CCTs for each thread in a program execution. Most of CCTLib's data structures are thread local except `IPNode`s, which are allocated from a shared memory pool. The allocation of `IPNode`s is non-blocking since it uses atomic fetch-and-add operations. Consequently, multiple threads can build/manage their own CCTs concurrently. Since threads can have parent-child relationships, we associate the root of the child thread's CCT to the creation point of its parent's CCT.

Since Pin does not provide information about parent-child relationships among threads, CCTLib performs extra work to establish this relationship. We override `pthread_create` to accomplish this. Immediately after a parent thread returns from `pthread_create`, CCTLib publishes its call-path and waits for the analysis routine added as part of a new thread creation to execute. During this interval, all threads attempting to spawn children thread(s) are paused from making progress. The analysis routine executed as part of the newly spawned thread attaches the published call-path as its parent, after which all threads resume.

6.6.1.9 Current limitations

Our current implementation does not support attaching to a running process. While we can determine partial call-paths seen after attaching to a running process, obtaining parent-child thread relationships and handling exceptional flows of control, are problematic. We wish to address this in future work.

6.6.2 Data-centric attribution in CCTLib

CCTLib supports attribution of memory addresses to heap and static data objects. For stack data objects, CCTLib marks them as on stack but does not associate them with individual stack variables. It is straightforward to capture the memory range for each thread's stack and eliminate addresses falling in that range from further analysis.

6.6.2.1 Instrumentation for heap data objects

CCTLib instruments memory management functions such as `malloc`, `calloc`, `realloc`, and `free`. CCTLib uses the calling context of the dynamic memory allocation site such as `malloc` to uniquely identify a heap data object. The memory range and its associated allocation call-path identifier are maintained in a map for future attribution of metrics to data objects.

6.6.2.2 Instrumentation for static data objects

CCTLib uses symbol names to uniquely identify static data objects. CCTLib reads the symbol table from each loaded module and extracts the name and memory range allocated for each static data object. It records this information in a map for each load module.

6.6.2.3 Associating address to data object at runtime

A CCTLib client Pin tool queries `GetDataObjectHandle`, passing it an effective address. The `GetDataObjectHandle` API returns a `DataHandle_t`, which is a 40-bit handle that uniquely represents the data object. Eight bits are reserved to distinguish variable types—stack, heap, and static. The remaining 32 bits represent the object. For heap objects, the 32-bit handle is a CCT handle (`ContextHandle_t`). For static variables, the handle is an index into a string pool of symbol names.

Aforementioned maps for associating addresses to objects should allow concurrent queries by multiple threads to ensure scalability. We provide two implementations, one is parsimonious in memory but less efficient in performance, and another one requires large memory but delivers efficient, constant-time lookup.

The first approach uses a novel balanced-tree-based map data structure that allows concurrent reads while the map is being updated. There is one map per load module and one map for heap addresses. <Address range, object handle> pairs are recorded in sorted order in these maps. Lookup in the tree has logarithmic cost. While map lookup operations are frequent, insertion and deletion are not. We devised a data structure where insertion and deletion are serialized but address lookup operations are fully concurrent. We accomplish this by maintaining a replicated tree data structure, where all readers are in one tree and writes update the other tree. The trees are swapped after a phase of updating by the writers. The result delivers scalable high performance for threaded programs.

The second approach employs a page-table based shadow memory analogous to [42]. For every allocated byte in the program, the shadow memory holds its 40-bit data handle. On each dynamic allocation, the corresponding range of shadow memory is populated with the call-path handle. For each symbol for static variables, the corresponding range of shadow memory is initialized with the string pool handle for the symbol. At runtime, for every accessed memory address, CCTLib maps the address to its shadow location and fetches the corresponding data handle in constant time. The shadow memory lookup and update operations need no locking; hence the approach scales well. Naturally, shadow memory introduces at least 5× memory bloat.

CCTLib provides both balanced-tree-based and shadow-memory-based methods for its client tools to trade off memory consumption vs. runtime overhead. In the next section, we will compare the performance of these different approaches.

6.7 Evaluation

In this section, we evaluate CCTLib's runtime and memory overhead on serial applications, as well as its scalability on parallel programs. We conducted our single-threaded experiments on 16-core `Intel Sandy Bridge` machines clocked at `2.2GHz`, with 128GB of 1333MHz DDR3 running `Red Hat Enterprise Linux Server v6.2` and `GNU 4.4.6` tool chain. We conducted our parallel scaling experiments on a quad-socket 48-core system with four `AMD Opteron 6168` processors clocked at `1.9 GHz` with 128GB of 1333MHz DDR3 running `CentOS 5.5` and `GNU 4.4.5` tool chain. All applications were compiled with `-O2`

optimization. We chose a subset of the SPEC CPU2006 integer reference benchmarks [85] along with three other applications for evaluation. Each application was chosen to represent a variety.

- ROSE compiler [190] : ROSE source-to-source compiler has close to 3 million lines of C++ code. We applied CCTLib when ROSE was compiling 80K lines of its own header files. The code does not show any spatial or temporal data locality. It has a large code footprint and deep call chains.

- LAMMPS [189]: LAMMPS is a molecular dynamics code with 500K lines of C++ code. We applied CCTLib on the `in.rhodo` input, which simulates Rhodo spin model. The code is compute intensive and has deep call chains.

- LULESH [112]: is a shock hydrodynamics mini-app; it solves the Sedov Blast Wave problem, which is one of the five challenge problems in the DARPA UHPC program. The code is memory intensive and does not have good parallel efficiency. It has frequent memory allocations and deallocations, which makes it an interesting use case for data-centric analysis.

6.7.1 Runtime overhead on serial codes

Table 6.3 shows runtime overhead for executions of the benchmark codes *compared to an unmonitored program.* For SPEC CPU2006 benchmarks that have multiple inputs, we show the mean values (arithmetic mean for raw values and geometric mean for relative quantities) across all inputs. Column 2 shows the running time for each program in seconds. `h264ref` program for the input `sss_encoder_main.cfg` (not shown) with 618 seconds of original execution time was the longest running program.

Columns 3–6 focus on CCTLib's call-path collection overhead. Since FEMs that track each memory access instruction are very common, we provide CCTLib's overhead for call-path collection on each memory access instruction in column 3, which has a modest average overhead of 19×. Column 4 shows CCTLib's overhead for collecting the call-path on each machine instruction, which has average overhead of 30×. Column 5 shows DeadSpy's overhead for collecting call-path on each machine instruction. For comparison purposes, we

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7
		Context-centric analysis				
Program	Original run time in sec	CCTLib overhead tracking memory access instructions	CCTLib overhead tracking each instruction	DeadSpy overhead tracking each instruction	CCTLib improvement over DeadSpy	CCTLib data-centric analysis overhead
astar	276.26	14×	22×	32×	32%	28×
bzip2	111.71	19×	32×	45×	28%	42×
gcc	44.61	23×	35×	54×	35%	44×
h264ref	260.12	31×	48×	72×	33%	67×
hmmer	326.32	21×	30×	42×	28%	47×
libquantum	462.38	22×	39×	73×	47%	46×
mcf	319.97	6×	10×	14×	28%	15×
omnetpp	352.30	14×	23×	35×	35%	34×
Xalan	294.80	32×	50×	78×	36%	65×
ROSE	23.64	30×	41×	53×	23%	49×
LAMMPS	99.28	17×	29×	39×	27%	40×
LULESH	67.29	20×	36×	46×	22%	48×
GeoMean	-	19×	30×	45×	31%	41×

Table 6.3 : Runtime overhead of CCTLib.

extracted DeadSpy's CCT construction and adapted it to gather the call-path on each instruction without performing dead write detection. Column 6 shows CCTLib's improvement over DeadSpy; CCTLib is about 30% faster than DeadSpy on average. The average difference in overhead is about 14× between CCTLib and DeadSpy. As stated before, CCTLib produces call site level attribution at all levels of call-path, which DeadSpy cannot.

Finally, column 7 shows CCTLib's overhead for performing data-centric analysis on each memory access besides performing call-path collection on each instruction. The values in column 7 subsume the overhead in column 4. We employed the shadow memory technique for data-centric analysis in these experiments. The average overhead for data-centric and context-centric attribution together is 41×.

The intended use of CCTLib is in conjunction with a Pin tool. Hence, the slowdown introduced by CCTLib atop a baseline Pin tool is of more interest to a tool writer. Table 6.4 shows the runtime overhead for executions of our benchmark codes *compared to a simple Pin tool that counts the number of instructions executed*. While one can write a smarter instruction counting tool, we chose a simple implementation that increments a counter before executing every instruction. Such instrumentation is representative of sophisticated tools that perform analysis at instruction-level granularity (e.g., dynamic data race detection and runtime computational redundancy detection). Column #2 shows that on average CCTLib

Column 1	Column 2	Column 3	Column 4
	Overhead of CCTLib wrt an instruction counting Pin tool		
	Calling context on	Data-centric analysis on each instruction	
Benchmark	each instruction	Tree-based	Shadow memory
astar	1.59×	4.01×	2.09×
bzip2	1.89×	4.04×	2.45×
gcc	2.07×	4.86×	3.01×
h264ref	1.35×	4.01×	1.87×
hmmer	0.87×	2.97×	1.33×
libquantum	2.79×	4.49×	3.15×
mcf	1.77×	4.91×	2.57×
omnetpp	1.76×	7.71×	2.54×
Xalan	2.37×	6.77×	3.07×
ROSE	1.43×	3.59×	1.75×
LAMMPS	1.79×	4.84×	2.47×
LULESH	1.77×	3.57×	2.49×
GeoMean	1.72×	4.49×	2.33×

Table 6.4 : CCTLib overhead compared to a Pin tool's overhead

introduces only 1.72× overhead atop a tool that counts every instruction. Column #3 shows that on average CCTLib introduces 4.45× overhead for performing data-centric attribution via the balanced-tree-based technique. Column #4 shows that on average CCTLib introduces 2.33× overhead for performing data-centric attribution via the shadow-memory-based technique. The relative overheads of CCTLib shown in Table 6.4 are near its worst-case performance since the work performed in the analysis routine of a simple instruction counting Pin tool is negligible (one addition operation). The overhead of CCTLib is independent of the overhead of a Pin tool's analysis function. Hence, for Pin tools that perform progressively more amount of work in their analysis routines (e.g., a data-race detector) the relative overhead of CCTLib progressively diminishes.

6.7.2 Memory overhead on serial codes

Table 6.5 shows the results of memory overhead introduced by CCTLib. Column 3 shows the maximum number of call-paths collected in each application. CCTLib collected ∼1.49 billion call-paths during an execution of the ROSE compiler. When tracking memory access instructions, CCTLib requires 3.49× extra memory on average for storing the CCTs. When tracking all instructions, CCTLib requires 4× extra memory on average. When performing data-centric analysis via balanced-tree-based technique, CCTLib requires 4.38× extra memory on average. When performing data-centric analysis via shadow-memory-based technique,

Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7
	Original	Max	**Contex-centric analysis**		**Data-centric analysis**	
Program	resident memory in MB	call-paths	Overhead tracking memory access instructions	Overhead tracking each instruction	Overhead (tree-based)	Overhead (shadow memory)
astar	230	3.04E+05	1.16×	1.17×	1.34×	8.65×
bzip2	561	8.12E+04	1.11×	1.12×	1.12×	8.03×
gcc	453	8.16E+08	15.6×	26.0×	26.0×	36.4×
h264ref	37	1.46E+06	2.49×	2.69×	2.91×	11.5×
hmmer	15	2.50E+05	4.38×	4.36×	5.13×	29.4×
libquantum	96	1.42E+05	1.28×	1.30×	1.32×	11.9×
mcf	1677	7.99E+05	1.02×	1.03×	1.03×	6.53×
omnetpp	170	8.67E+06	1.87×	2.35×	3.76×	10.5×
Xalan	419	6.58E+08	25.9×	38.6×	39.1×	46.6×
ROSE	380	1.49E+09	64.9×	98.2×	100×	105×
LAMMPS	110	1.23E+06	1.58×	1.57×	1.70×	16.9×
LULESH	26	2.84E+05	2.28×	2.27×	2.51×	9.11×
GeoMean	-	-	3.49×	4.00×	4.38×	16.9×

Table 6.5 : Memory overhead of CCTLib.

CCTLib requires 16.9× extra memory on average. Column 6 and column 7 subsume the overhead in column 5 since they include the overhead of CCT construction for each instruction.

There is a correlation between the number of call-paths collected and the memory overhead of call-path collection. Applications such as `gcc`, `Xalan`, and `ROSE` have deep recursion leading to same instructions being repeated in multiple calling contexts, consequently they have a higher memory overhead. If we ignore `gcc`, `Xalan`, and `ROSE`, the geometric mean for columns $4, 5, 6$, and 7 are 1.71×, 1.77×, 1.99×, and 11.4×, respectively.

Our 40-bit `DataHandle_t` structures are word aligned to 64 bits, causing about 8× memory overhead for shadow-memory-based data-centric analysis. One can use packed structures and trade off memory for runtime overhead. We did not explore that option in our evaluation. For the `hmmer` benchmark, the memory overhead is higher for shadow-memory-based data-centric analysis; this is because the application has a relatively smaller memory footprint (15MB), whereas CCTLib preallocates a page directory of 8MB for its two-level page tables to maintain the shadow memory. This 8MB of CCTLib's memory, in comparison with 15MB of the application's working set, skews the numbers.

6.7.3 Scalability on parallel applications

To evaluate CCTLib's scalability, we perform strong scaling experiments with the OpenMP versions of LAMMPS and LULESH. We varied the number of threads from 1 to 32 and evaluated the slowdown caused by CCTLib when compared to the same configuration without monitoring. We performed our experiments with call-path collection as well as with data-centric attribution. For data-centric attribution, we tried both balanced-tree-based implementation and shadow-memory-based implementation.

We define $H(n)$—the overhead in an n-thread execution, as the ratio of $R_c(n)$—the running time of CCTLib in an n-thread execution, to $R_o(n)$—the running time of the original n-thread execution. We define $S(n)$—the percent parallel scalability of n-thread execution of CCTLib, as a scaled ratio of the overhead of a 1-thread execution to the overhead of an n-thread execution.

$$H(n) = \frac{R_c(n)}{R_o(n)}$$
$$S(n) = \frac{H(1)}{H(n)} \times 100$$

High scalability is best. The results are tabulated in Table 6.6 and Table 6.7 for LAMMPS and LULESH, respectively.

We make the following observation about CCTLib:

- For call-path collection, the overhead remains fairly stable (22×–27×) for LAMMPS and achieves 96% scalability. For LULESH, the overhead decreases dramatically with increased parallelism. This behavior is because of the Amdahl's law, since LULESH has poor parallel efficiency, injecting the scalable CCTLib component into it causes CCTLib's overhead to scale super-linearly (586%).

- Data-centric attribution using our concurrent balanced-tree-based technique has higher overhead for LAMMPS (about 80×); however, the overhead remains stable and achieves perfect scaling (96%). For LULESH, the overhead steadily reduces with increase in the number of threads, reaching 607% scalability at 32 threads. Since the balanced-tree-based technique has a lower memory footprint, it might be preferred

		LAMMPS no. threads					
		1	2	4	8	16	32
Original running time in seconds		173.73	95.32	53.43	30.81	21.35	17.85
Original prog. parallel efficiency		100%	91%	81%	70%	51%	30%
Call-path analysis	Overhead	22×	22×	23×	27×	24×	23×
	Scalability	100%	99%	96%	82%	93%	96%
balanced-tree-based data-centric analysis	Overhead	80×	78×	79×	83×	85×	80×
	Scalability	100%	102%	100%	96%	94%	100%
Shadow-memory-based data-centric analysis	Overhead	32×	31×	32×	36×	38×	34×
	Scalability	100%	102%	100%	90%	84%	95%

Table 6.6 : Parallel efficiency of CCTLib on LAMMPS.

		LULESH no. threads					
		1	2	4	8	16	32
Original running time in seconds		137.29	86.30	47.82	28.75	29.99	61.37
Original prog. parallel efficiency		100%	80%	72%	60%	29%	7%
Call-path analysis	Overhead	21×	18×	18×	17×	9×	4×
	Scalability	100%	120%	116%	125%	229%	586%
balanced-tree-based data-centric analysis	Overhead	49×	45×	44×	39×	22×	7×
	Scalability	100%	109%	111%	124%	228%	670%
Shadow-memory-based data-centric analysis	Overhead	33×	28×	32×	33×	23×	8×
	Scalability	100%	115%	101%	100%	143%	408%

Table 6.7 : Parallel efficiency of CCTLib on LULESH.

over our shadow-memory-based technique when the concurrency level is higher and the application has poor scalability.

- Data-centric attribution using our shadow-memory-based technique is significantly faster than our balanced-tree-based technique and it scales well. On LAMMPS, the technique introduces 31×–38× slowdown and shows 95% scalability. For LULESH the trend is similar to that of the call-path collection, lowering overhead to 8× with the increase in the number of threads causing it to attain 408% scalability.

6.8 Discussion

Attributing metrics to code for every executed instruction was considered infeasible for long running programs due to the purported space and time overheads [69, 229]. In this chapter, we debunked this myth. We demonstrated that, with CCTLib, one can collect call-paths and attribute memory accesses to data objects on every executed machine instruction, even for reasonably long running programs. CCTLib has modest runtime overhead and scales

well when used on multithreaded codes. Our choice of algorithms and data structures within CCTLib makes such a heavyweight task affordable, efficient, and scalable.

Fine-grained execution monitoring tools are of critical importance in performance analysis and correctness tools, among others. An average overhead of 1.72× compared to a simple Pin tool to associate every instruction to its calling context is quite affordable. Since the overhead of CCTLib is constant, the relative overhead of CCTLib will be minuscule when used in conjunction with heavyweight analysis tools such as a data race detector that incur several orders of magnitude overhead. We believe CCTLib provides a framework that will allow many Pin tools to provide rich, diagnostic, contextual information along with their analyses.

The CCTLib framework has attracted interest. Intel's PinPlay [186] and DrDebug [235] are some of the candidate Pin tools interested in employing CCTLib. Other researchers have started building tools that use CCTLib for dynamic detection of computational redundancies [236], associating memory footprint to calling contexts, among others.

Chapter 7

Related Work

Art is essentially communication. It doesn't exist in a vacuum. That's why people make art, so other people can relate to it.

Conor Oberst

We present the related work pertaining to the areas of GPU performance analysis, synchronization optimization, shared-memory mutual exclusion, redundancy elimination, call-path collection, and data-centric attribution in Sections 7.1-7.5 respectively.

7.1 GPU performance analysis

Several strategies have been proposed [187, 213] that assess the performance of GPU kernels alone. Less, however, has been done to provide holistic performance analysis of hybrid parallel applications on heterogeneous supercomputers. In Section 7.1.1-7.1.3 we discuss the related work in GPU-kernel performance analysis, system-wide performance analysis, and root-cause performance analysis, respectively.

7.1.1 GPU-kernel performance analysis

With regard to performance tools focused exclusively on GPU kernels, NVIDIA provides a state-of-the-art measurement-based tool—*Nvidia Visual Profiler (NVP)* [177]. NVP traces the execution of each GPU task, recording method name, start and end times, launch parameters, and GPU hardware counter values, among other information. NVP provides an extension called NVTX that allows programmers to manually instrument CPU-side events. Other tools, e.g., [14, 91, 213], employ modeling and/or simulation to provide insight into

the performance of individual kernels.

Although these kernel-focused tools are *aware* of calls to CUDA host APIs on the CPU, none of these tools provides significant insight into CPU activity of heterogeneous applications. Furthermore, most of these tools cannot gather performance of concurrent executions of multiple GPU kernels; instead, they serialize GPU kernels as well as CPU threads. Such intrusive analysis, while useful in characterizing the behavior of a stand-alone kernel, is often undesirable in large CPU-GPU applications that employ a high amount of asynchrony.

7.1.2 System-wide performance analysis

With regard to performance tools focused on both GPU and CPU activities, two leading tools are TAU [148] and VampirTrace [81]. Both of these tools offer an array of techniques for performance analysis of hybrid architectures.

While TAU has some support for sampling, it principally employs instrumentation for measuring performance. Consequently, TAU has a natural integration path to absorb GPU instrumentation. On Nvidia devices, TAU uses the CUPTI interface [173] to both monitor execution of GPU tasks and capture GPU hardware counter values. TAU considers code performance from multiple perspectives, including inter-node communication, intra-node execution, CPU-GPU interactions, and GPU kernel executions.

The Vampir performance analysis toolset [81] traces program executions on heterogeneous clusters. VampirTrace monitors GPU tasks using CUPTI, logging information including kernel launch parameters, hardware counter values, and details about memory allocations. For monitoring the CPU activity, the Vampir toolset collects a trace of function entry/exit like TAU. Vampir toolset performance analysis of hybrid codes is based on post-mortem analysis of traces.

TAU and Vampir toolsets do not offer any automatic techniques for idleness analysis, although one may indirectly infer idleness via manual inspection of their profiles or traces.

7.1.3 Root-cause performance analysis

Curtsinger and Berger [49] have proposed "causal profiling" as a technique for programmers on where to focus their optimization efforts. Causal profiling measures the impact of speeding

up a code on the overall execution time. One cannot speed-up a code automatically without optimizing it. One can, however, slow a code down by artificially injecting pauses. If slowing down a code slows down the entire execution, then such code is on the critical path and deserves programmer's attention. With the guidance from causal profiling, they improve the performance of Memcached [66] by 9%, SQLite [197] by 25%, and accelerate six PARSEC [19] applications by up to 68%.

Tallent et al. [225] developed *directed blame shifting* in the context of lock waiting, where lock-waiting threads blame a single lock holder. Tallent and Mellor-Crummey [222] developed *undirected blame shifting* in the context of work stealing runtime such as Cilk, where some threads may be working while others may be idle. The undirected blame shifting apportions the blame among multiple working threads, unlike the directed blaming, which targets one thread. Liu et al. [137] have developed a blame shifting mechanism for attributing idleness and lock waiting to identify in OpenMP codes. They classify execution of OpenMP programs into *work, idleness, overhead,* and *lock waiting*. Since the number of threads in OpenMP can dynamically change, they effectively integrate the *directed blame shifting* with *undirected blame shifting*. CPU-GPU blame shifting with multiple GPU streams and CPU threads, presented in Chapter 2, bears an analogy with Liu et al.'s blame apportioning techniques. With the insights gained from blame shifting, they tune several OpenMP applications and accelerate them by up to 82%.

The CPU-GPU stall analysis, presented in Chapter 2, resembles the root-cause analysis capability in Scalasca [22]. Scalasca employs source instrumentation to trace communication events and performs forward and backward replay of traces to associate the cost of wait states to their causes. Our technique differs from Scalasca by using sampling-based measurement. Our technique identifies one specific case of waiting that is caused by blocking system calls. Our technique identifies the impact of late arrivers in only collective communications whereas Scalasca is more generic and identifies causes of delays in both point-to-point and collective operations.

Recently, Schmitt et al. [203] developed a tool for identifying critical optimization targets in heterogeneous applications. They extend Bohme et al.'s [22] work on identifying the root cause of wait states in MPI programs and our work on CPU-GPU blame shifting. They

develop *critical blame* scheme, a combination of critical-path and root-cause analysis, which is aware of three important programming models OpenMP, CUDA, and MPI. During a postmortem analysis pass of the execution traces, the *critical blame* scheme considers both the critical path as well as the assigned blame to rank order a code region as a potential target for optimization. They employ program instrumentation to gather traces.

7.2 Synchronization optimization

In Section 7.2.1 and 7.2.2, we discuss the related work in static and dynamic analysis for synchronization optimization, respectively. In Section 7.2.3, we discuss the related work in lightweight call-path collection to draw an analogy with our work in context collection used for the barrier elision work.

7.2.1 Static analysis

Static program analysis that examines the synchronization in parallel programs has been applied for performance optimizations. Concurrency analysis can be traced back to the work of Shasha and Snir [210]; they build a set of models that determines when consecutive operations in a program segment of a parallel program may execute concurrently, without violating the programmer's view. Jeremiassen and Eggers [105] present a static barrier analysis for SPMD codes (Stanford SPLASH) used to eliminate false sharing on shared memory machines. Zhang et al. [245, 246] present concurrency analyses for shared (OpenMP) and distributed (MPI) programming models with textually unaligned barriers. Kamil and Yelick [109] describe a static data race detection in the Titanium programming language that employs textually aligned barriers. Dotsenko [67] developed an algorithm for synchronization strength reduction (SSR), which replaces textual barriers with non-blocking point-to-point synchronization. Agarwal et al. [3] present a may-happen-in-parallel analysis for X10 programs with unstructured parallelism.

7.2.2 Dynamic analysis

Runtime elision of synchronization operations has received a fair share of attention in both software and hardware. Many techniques have been developed for lock elision in Java, `glibc`, or the Linux kernel. At the hardware level, speculative lock elision described by Rajwar and Goodman [192] is available now in hardware implementations such as Intel TSX. We discuss mechanisms for eliding mutual exclusion more in detail in Section 7.3.7. None of these mechanisms, however, elide collective communication such as barriers.

Sharma et al. [209] describe a dynamic program analysis to detect functionally irrelevant barriers in MPI programs. In their definition, a barrier is irrelevant if its removal does not alter the overall MPI communication structure of the program. They use a model checker to try program interleavings. Their technique uncovers irrelevant barriers in many small benchmarks containing a few hundred lines of code.

An orthogonal approach to our automatic elision is profiling for redundant barriers followed by code modification to insert additional arguments (similar to C++ default arguments) at call sites leading to redundant barriers. The call-sites can use the runtime call-path prefix to recognize barrier redundancy. The guards would take the form of a boolean flag indicating the necessity or redundancy of a barrier for the downstream APIs. We are unaware of any prior work analogous to this idea.

7.2.3 Lightweight call-path collection

Lightweight instrumentation of parallel programs is a well-explored area. Library interposing and link-time function wrapping are standard techniques to intercept function calls. These techniques are extensively used in performance analysis tools such as HPCToolkit [224]. Our on-demand call-stack-unwinding technique on each barrier bears similarity with HPCToolkit's call stack sampling. HPCToolkit maintains an unwind recipe for each range of instructions by analyzing the binary code at runtime. Our barrier elision technique deviates from this since we compile the code with frame pointers to ensure perfect stack unwinds. The binary analysis is directly applicable to improve our technique. An alternative context-collection technique is instrumenting every call and return instruction, and eagerly com-

puting the call-path [41] in a "shadow" stack. Eager call-path collection techniques, while suitable for frequent unwinds, are unsuitable when the call-path is infrequently needed. Context-sensitive, dynamic analyses have been explored in computer security also. Most of the approaches use stack walking for context identification. Recent work by Bond and McKinley [25, 26] refines the notion of a context to avoid stack-unwinding overhead. Their probabilistic calling contexts are directly usable in the calling context used by our barrier elision scheme.

7.3 Shared-memory mutual exclusion algorithms

Algorithms for shared-memory mutual exclusion have been extensively studied. Mellor-Crummey and Scott [157] provide an extensive empirical and theoretical analysis of shared memory synchronization mechanisms available in the early 1990s. We discuss various kinds of locks relevant to our work in Section 7.3.1-7.3.4. We discuss the related work in the area of the fast-path in Section 7.3.5. We discuss the related work in the area of software-based contention management in Section 7.3.6. We discuss the related work in the area of hardware transactional memory Section 7.3.7. We discuss the related work in the empirical and analytical analysis of locks in Section 7.3.8 and 7.3.9, respectively.

7.3.1 Queuing locks

Queuing locks enqueue lock requests in a FIFO queue, and the waiting threads spin on unique locations until their predecessors signal them. MCS lock [157] and CLH lock [146] are two prominent state-of-the-art queuing locks.

The MCS lock acquire protocol enqueues a record in the queue for a lock by 1) swapping the queue's tail pointer with a pointer to its record and 2) linking behind a predecessor (if any). If a thread has a predecessor, it spins locally on a flag in its record. Releasing an MCS lock involves setting a successor's flag or resetting the queue's tail pointer to null if no successor is present.

The CLH lock [146] is analogous to the MCS lock, with the following two differences: 1) each thread enqueues its record into a queue when acquiring a lock but does not reclaim its

record on release; instead, its successor is responsible for its record, and 2) a thread waiting to acquire a lock waits on a flag in its predecessor's record rather than its own.

Our work is focused on mutual exclusion algorithms for NUMA architectures, which is a more recent architectural advancement, and hence, we cover only the related work in the area of locks for NUMA architectures.

7.3.2 Hierarchical locks

Radovic and Hagersten [191] designed a hierarchical back-off (HBO) lock. The idea of the HBO lock is simple: when the lock is acquired, the domain id of the acquiring thread is CASed into the lock variable. Other threads from the same domain, waiting to acquire the lock, spin with a smaller back-off value, whereas threads from a different domain spin with a larger back-off value. A thread sharing the same domain as the lock holder is more likely to subsequently acquire the lock when it is freed, compared to contending threads in other domains.

The HCLH lock [143] is a variant of the CLH lock that is tailored for 2-level NUMA systems. Threads on cores of the same chip form a local CLH queue. The thread at the head of the queue splices the local queue into the global queue. The splicing thread may have to wait for a long time or splice a very short queue of local threads, both of which lengthen the critical path. The HCLH lock can pass the lock within a domain only once before relinquishing the lock to another domain. Furthermore, the HCLH lock does not exploit the locality beyond two levels of a NUMA hierarchy.

The Cohort MCS (C-MCS[1]) lock by Dice et al. [63] for NUMA systems employs two levels of MCS locks, treating a system as a two-level NUMA hierarchy. One MCS lock is local to each NUMA domain and another MCS lock (global) is shared by domains, which protects the critical section. Each thread trying to enter a critical section acquires the local lock first. The first thread to acquire the local lock in each domain proceeds to compete for the global lock while other threads in each domain spin wait for their local lock. A releasing thread grants the global lock to a local successor by releasing the local lock. A

[1]Dice et al. refer to thier lock as C-MCS-MCS, we call it as C-MCS for brevity.

lock passes within the same domain for a *threshold* number of times at which point the domain relinquishes the global lock to another domain. Dice et al. also explore other lock cohorting variants beyond C-MCS that employ various locking protocols at each of the two levels, with the local and global locking protocols selected independently. We focused on the MCS lock at each level of the HMCS lock leaving exploring different locks at different levels for the future work. Moreover, even Dice et al.'s best-performing back-off and MCS lock combination (C-BO-MCS) is only marginally better than the second best C-MCS lock. Furthermore, the C-BO-MCS lock has unbounded unfairness, whereas the C-MCS lock has bounded unfairness. Cohort locks do not exploit the locality beyond two levels of the NUMA hierarchy.

7.3.3 Combining locks

The "combining" conceptually batches tasks from multiple threads into one thread.

Oyama et al. [182] devised a "combining" lock that delegates the critical section of all waiting threads to a lock holder. In their design, the `lock` variable assumes three values—`locked`, `free`, and `conflict`. When the `lock` is `free`, a thread—wanting to acquire the lock—CASes it to `locked`; if the CAS succeeds, then the thread becomes the lock holder and enters the critical section. All threads that fail to CAS the `lock` variable, prepare a record containing the continuation context needed to execute their critical sections and atomically push a pointer to their record on to a stack of such records. In fact, the top of the stack is the same `lock` variable, if its value is neither `locked` nor `free` (i.e., `conflict`). If there is a set of stacked contexts when a lock holder is about to release the lock (the `lock` points to `conflict`), then the lock holder detaches the contexts and executes each thread's critical section in the detached context. Eventually, the lock holder either releases the lock (CASes it to `free`) or begins working on the new set of accumulated contexts (if any). By delegating the critical section of many threads to a lock holder, this lock design exploits the locality of data accessed in the critical section. Since the waiting threads are arranged in a LIFO data structure, this lock is particularly unfair. In addition, this lock design can cause starvation for the lock holder thread by not allowing it to finish its release protocol.

Fatourou and Kallimanis [72] revised the combining lock of Oyama et al. They propose

two locks: CC-Synch for cache-coherent machines and DSM-Synch for machines without caches. Conceptually, CC-Synch implements a "combining-friendly" CLH lock so that the combiner thread can traverse the queue. Each record in the queue contains the context necessary for a combiner thread to execute another thread's critical section. The thread that is assigned the head of the list plays the role of the combiner. The combiner begins by serving its request first. Other threads that have announced their requests, spin on a unique field of in their predecessor's record. The combiner does not release the lock after the completion of its critical section; instead, it continues serving the requests announced by the other threads, followed by indicating the completion of requests to the waiting threads. The CC-Synch protocol places a bound on the number of critical sections a combiner thread services. Having a bound ensures starvation freedom. Since CC-Synch is modeled after the CLH lock, a waiting thread spins on a field in its predecessor record, which generates an unbounded number of remote memory requests on machines that do not have caches. DSM-Synch lock is modeled after the MCS lock, where each thread spins on a node that is locally allocated; this design eliminates the unbounded remote memory references.

Dice et al. devised the flat-combining MCS lock [62] (FC-MCS), which combines the flat-combining synchronization paradigm [84] and the MCS lock algorithm [157]. The key idea of flat combining (FC) is to implement a concurrent data structure given its sequential counterpart. The FC technique employs mutual exclusion to pick repeatedly a unique "combiner" thread that will apply all other threads' operations to the structure. The key advantage is that when one thread applies n operations consecutively on a shared data structure instead of n threads applying one operation each, data locality is maintained and synchronization overhead is reduced. The FC-MCS builds local queues of waiting threads and employs a designated "combiner" thread to join local queues into a global MCS queue. The FC-MCS lock can pass the lock within a domain only once before relinquishing the lock to another domain (if present). Furthermore, the FC-MCS lock does not guarantee the locality beyond two levels of the NUMA hierarchy.

Lublinerman et al. [142] developed "delegated isolation" in the context of the Habanero Java parallel programming language [37]. Delegated isolation, conceptually, allows threads to launch their mutually exclusive regions of arbitrary granularity as separate tasks. Each

task owns a region of data. When a task T_1 needs to own a datum owned by another task T_2, T_1 delegates its work, giving the ownership of its data to T_2. T_2 takes the responsibility of "re-executing" the tasks of T_1. By delegating the work of accessing the shared data to a thread already owning such data, the approach can reduce contention. In addition, delegated isolations provide dynamism, safety and liveness guarantees, and programmability when handling mutual exclusion.

7.3.4 Dedicated server threads for locks

Lozi et al. [140] have devised an effective mechanism to address NUMA effects via the Remote Core Locking (RCL) technique. The RCL locking adopts a client-server model for executing critical sections. RCL dedicates one or more "server" threads that execute the critical sections. Several server threads, typically, share a single hardware thread. By creating more than one server thread, on demand, the RCL lock provides the liveness and responsiveness, which are needed when a server thread blocks inside a critical section, encounters an O/S preemption, or busy waits on a flag. Typically, a single hardware thread accesses the data inside any critical section, which ensures the locality of data on NUMA machines. The lock data structures used in delegating the work from a client thread to a dedicated thread, however, do not have locality, which is one of the limitations of the RCL lock compared to the HMCS lock. Furthermore, RCL lock dedicates at least one hardware thread (and several logical threads). An RCL lock may introduce artificial serialization of a parallel program since an RCL server thread executes all critical sections serially even if they are protected by different locks. Nevertheless, using RCL locks, the authors demonstrate performance improvements of up to 2.6× with respect to POSIX locks on Memcached, and up to 14× with respect to Berkeley DB.

Table 7.1 summarizes various features of the HCLH, C-MCS, FC-MCS, RCL, and FP-AHMCS locks.

7.3.5 Fast-path techniques for mutual exclusion

Yang and Anderson [240] have devised an N-process software-based lock mechanism that only requires read and write operations. The key idea in this lock design is to use a binary

Feature	H-CLH	C-MCS	FC-MCS	RCL	FP-AHMCS
Has all levels of locality of lock data structures?	No	No	No	No	Yes
Has all levels of locality of the critical section data?	No	No	No	Yes	Yes
Can be tuned for throughput?	No	Yes	No	N/A	Yes
Can be tuned for fairness?	No	Yes	No	N/A	Yes
Has freedom from additional memory management?	No	No	No	Yes	Yes
Has freedom from dedicated resources (thread)?	Yes	Yes	Yes	No	Yes
Suitable under low contention?	No	No	No	No	Yes

Table 7.1 : Comparison of various NUMA-aware locks.

arbitration tree. The lock acquisition begins at the leaf of the tree for each thread and proceeds through the spine of the tree to the tree root. Only two threads contend at each tree node. One thread gets blocked and busy waits while the other thread proceeds to the parent node in the tree. The thread reaching the root of the tree can enter its critical section. After executing it critical section, the lock holder traverses the spine of the tree from the root to its leaf, each time unblocking a busy-waiting thread at each level. The HMCS lock also employs an arbitration tree, but the HMCS tree is an n-ary tree, and the arity of any tree node can be arbitrary. The HMCS lock differs from the Yang and Anderson's lock in two key ways: 1) each thread need not acquire all the locks from leaf to the root each time to enter the critical section 2) the lock releasing thread does not traverse the tree from root to leaf, instead, it passes the lock to a waiting thread sharing the lowest common ancestor. Because of these differences, while the HMCS lock enhances locality, the Yang and Anderson's lock negates locality. Yang and Anderson's lock's time complexity is $O(\log_2 N)$, where N is the number of participating threads.

The FP-HMCS lock, presented in Chapter 4, is inspired by the fast-path slow-path technique devised by Lamport [124] and adapted by Yang and Anderson in their binary-arbitration-tree-based lock. Yang and Anderson's initial version of the fast-path to their binary-arbitration-tree-based lock [240] overlays a 2-process mutual exclusion on top of the tree-based algorithm. When a process detects no contention, it directly contends for the top-level 2-process mutual exclusion, which is the fast-path. When a process detects contention, it follows the slow-path through each level of the tree. With this technique, the fast-path technique allows a process to enter its critical section in $O(1)$ time when there is no contention, and the slow-path allows a process to enter its critical section in $O(log_2 N)$ time.

Once the fast-path has been acquired, it is "closed" so that other processes do not simultaneously acquire it. Once a period of contention ends, the fast-path must be "reopened" for subsequent fast-path acquisitions. To reopen the fast-path, a process checks a flag in each process, to identify whether a process is still contending. This polling causes additional overhead in the release protocol making the algorithm's worst-case time complexity $\theta(N)$. In their improved fast-path mechanism for mutual exclusion [10] Anderson and Kim address the $\theta(N)$ time complexity by bounding it to $O(1)$ under no contention and $O(\log_2 N)$ under contention. Anderson and Kim's algorithm uses a tuple of boolean along with a process identifiers to indicate other processes, which process obtained the fast-path. In addition, they use an array of N auxiliary variables to rename process identifiers to distinguish a previously accessed fast-path from a new one. To bound the renaming to within N auxiliary variables, they recycle the identifiers using the modulo operation. When recycling is not safe, they deflect the process asking for the fast-path to take the slow-path, which happens only during contention, and yet maintains the $O(\log_2 N)$ time complexity invariant.

Kogan and Petrank [120] employ a *slow-path fast-path* technique to create fast wait-free data structures. In their approach, first they execute an efficient lock-free version of the algorithm for a bounded number of times, which is considered the *fast-path*. On failure to apply the operation via the lock-free version, the algorithm switches to the wait-free version, which guarantees completion in a bounded number of steps, which is considered the *slow-path*. They demonstrate the effectiveness of their approach by accelerating wait-free implementations of the concurrent queue and linked list implementations to match the performance with lock-free counterparts.

7.3.6 Software-based contention management

Reacting to contention in a system via software technique has been widely studied. Dice et al. [61] explore the effect of wrapping a CAS operation with software-based techniques such as constant back-off, exponential back-off, time slicing, MCS lock, and array-based lock to address high contention situations. They demonstrate that the software-based contention management improves the performance of common concurrent data structures such as queues and stacks. In the context of Software Transactional Memory (SMT), an array of contention

management techniques [77, 86, 94, 201, 218, 243] has been studied. A key distinguishing feature of adapting to contention described in this dissertation is that instead of consciously attempting to reduce the contention, our methods consider contention as an opportunity to exploit locality.

7.3.7 Hardware transactional memory for mutual exclusion

Hardware transactional memory (HTM) is an active research area for accomplishing synchronization with low overheads.

Rajwar and Goodman's "Speculative Lock Elision" [192] (marketed under the name "Hardware Lock Elision" (HLE) by Intel) is a technique to elide a write associated with a locking operation. The HLE introduces two special instruction prefixes `xacquire` and `xrelease`. Programmers can use the `xacquire` prefix before the lock acquiring instruction such as `xchg` and `xrelease` prefix before the matching lock release instruction. The hardware begins a transaction when it executes an `xacquire` instruction but hides the side effect of the memory update associated with the lock acquisition. Instead, the address of the lock is added to the read-set. Similarly, the hardware elides the memory update associated with the `xrelease` prefixed instruction if it were reverting the value of the location to the value seen before the start of the transaction. All instructions appearing between a pair of `xacquire` and `xrelease` prefixed instructions are executed as a transaction. If the lock was free before the `xacquire` prefixed instruction, all other processors continue to see it as available immediately after one or more processes execute an `xacquire` prefixed instruction. The hardware detects the conflicting accesses that occur during the transactional execution and aborts the transaction, if necessary. The processor attempts to commit the transactional execution on reaching the `xrelease` prefixed instruction. If multiple threads execute critical sections protected by the same lock without any conflicting data accesses, the threads can execute concurrently. If a transaction fails, the hardware executes the same region once again, this time non-transactionally and without eliding the `xacquire` and `xrelease` prefixed instructions.

Odaira et al. [178] eliminate the global interpreter locks in Ruby via hardware transactional memory. They dynamically adjust the transaction lengths on a per byte code basis

to reduce the chances of transaction aborts. They dynamically choose the best transaction granularity for any number of threads and applications that run for sufficiently long times. This technique of online learning about locks is similar to our adaptive HMCS scheme.

Rossbach et al. [199] explore the cooperation between locks and transactions in their TxLinux operating system. They introduce cooperative transactional spinlocks (cxspinlocks), which allow transactions and locks to protect the same data while maintaining the advantages of both synchronization primitives. Cxspinlocks allow TxLinux to attempt the execution of critical regions via transactions and abort the transaction if the region performs an I/O. If a transaction is aborted, the fallback mechanism uses the traditional spin locks.

GNU pthreads library has a prototype support for lock elision via HTM [116]. It introduces wrappers around the traditional `pthread_mutex_lock` and `pthread_mutex_unlock` routines. The acquire wrapper first starts a transaction and checks the lock variable adding it to its read set. If the lock is free, then the protocol returns, allowing the user code to execute the critical section as a transaction; otherwise, the protocol aborts the transactions and waits to acquire the lock by calling `pthread_mutex_lock`. Since the lock is added the transaction's read set, any change to the lock by another thread causes the transaction to abort. Similarly, conflicting accesses between two threads performing their transactions causes them to abort. If a transaction frequently runs for a long time before aborting, it can hurt the overall application performance. The elision wrapper uses an adaptive solution to handle frequent aborts. The algorithm remembers the code regions that frequently abort and temporarily disables such regions from taking the transactional path. Kleen [116] has identified that the behavior of the `pthread_mutex_trylock` API is different when it is nested inside a transaction vs. when it is nested inside a `pthread_mutex_lock`. Since a transaction does not take a lock, a `pthread_mutex_trylock` call succeeds when called from inside a transaction, whereas it fails when called from inside a `pthread_mutex_lock`.

7.3.8 Empirical evaluation of mutual exclusion techniques

Most recently, Tudor et al. [59] performed a detailed empirical evaluation of atomic operations (swap, compare-and-swap, fetch-and-add, test-and-set) and locks (test-and-set, test-and-test-and-set, ticket, CLH, and MCS) on a variety of hardware such as AMD Opteron, Intel

Xeon, Sun Niagara 2, and Tilera TILE-Gx36. Not surprisingly, they infer that crossing a socket is expensive. They do not perform any additional study to identify the cost of crossing a node. Our studies with SGI UV 1000 show that the locality is important not only within a socket but also at every level of the NUMA hierarchy—from within a core to within a rack of many nodes. They also notice that in directory-based architectures such as Opteron, sharing within the socket is necessary but not sufficient due to the broadcast messages that need to be issued when a cache line is shared. They notice that no single lock wins on all platforms. They make claims without a strong justification that 1) ticket locks are good enough, 2) simple locks are powerful under low contention, and 3) the scalability of synchronization is mainly a property of hardware. We claim that maintaining the locality of reference is not a property of hardware but that of the software running on it. We also claim that one can design a sophisticated lock with low overhead by ensuring that the complexity is not added to the critical path.

In the context of HTM, Yoo et al. [242] performed an empirical evaluation of Transactional Synchronization Extensions (TSX) on Intel architectures. They demonstrate the applicability (low overhead) of the Intel TSX on a CLOMP-TM [202], STAMP [159], and RMS-TM [114] benchmarks, a few real-world HPC programs, and a parallel user-level TCP/IP stack. They elide the locks with HTM inside the synchronization libraries used by the aforementioned codes. If the transactional execution fails repeatedly, then they fall back to explicitly acquiring the locks to ensure forward progress. They used a retry count of five before falling back to a lock-based mutual exclusion for their workloads. They hand optimize certain workloads by using lockset elision and transactional coarsening to achieve better performance. They observed a significant implementation challenge in porting codes with condition variables to work with HTM. From our experience porting a few applications to use the HMCS lock, we concur with the findings of Yoo et al. that the condition variables make the porting harder.

Nakaike et al. [163] compare the HTM implementation available in four hardware architectures: Blue Gene/Q, zEnterprise EC12, Intel Haswell, and POWER8 via the STAMP [159] benchmark. Nakaike et al. employ a more sophisticated strategy of retrying a transaction several times based on the failure code before falling back to taking a global lock; our

HTM-based locks can adopt their retry strategies. They demonstrate that no single HTM implementation is more scalable than the other in all of the benchmarks.

7.3.9 Analytical study of lock characteristics

There is limited prior work in analytical modeling of lock *throughput*. Boyd-Wickizer et al. [27] use Markov models for ticket-based spin locks to reason about an observed collapse in the performance of several threaded applications on Linux. We are unaware of any analytical modeling of lock *fairness*. Buhr et al. [33] conduct an empirical study of the fairness of various locks. To the best of our knowledge, this dissertation is the first to provide *combined throughput-fairness* analytical models for locks.

7.4 Redundancy elimination

Several compiler optimizations focus on reducing avoidable operations. Every execution speed related optimization tries to reduce operation count and/or memory accesses. An exhaustive review of compiler optimizations is beyond the scope of this proposal. We highlight some prior art related to our work.

Butts and Sohi [34] propose a hardware-based approach to detect useless operations and speculatively eliminate instructions from future executions by maintaining a history of instructions' uselessness. In SPEC CPU2000 integer benchmarks, they found on average 8.85% useless operations, and their technique shows an average speedup of 3.6%. Their work focuses on identifying and eliminating useless computations only; useless memory operations are never eliminated since mis-prediction can lead to the violation of memory consistency. Our approach is orthogonal to this; we do not track CPU-bound computations; instead, we track reads and writes only. The authors do not mention if they can provide any actionable feedback on the locations of useless computations whereas DeadSpy provides full context to take informed action.

Archambault [12] discusses an inter-procedural data-flow analysis technique to eliminate dead writes. In their patent, they suggest merging liveness information for global variables from basic blocks to create a set of *live on exit (LOE)* data structure for procedures. They

traverse the program call graph in depth-first order, propagating LOE through function calls and returns. However, they do not discuss the effectiveness of their technique. Being a static analysis technique, we suspect it has limitations when dealing with aggregate types, dynamic loading, and aliasing. In contrast, DeadSpy, which uses a dynamic technique, cannot distinguish between *dead for this program input* and *live for some other input*.

Gupta et al. [79] have proposed "predication-based sinking"—a cost-benefit based algorithm to move partially dead code from hot paths to infrequently executed regions. With profiling information, this technique can eliminate the intra-procedural dead writes.

The shadow memory technique, presented in our implementation, is used in several well-known tools such as *Eraser* [200] for data race detection, *TaintCheck* [170] for taint analysis, and *Memcheck* [207] for dangerous memory use detection. Nethercote et al. [167] present techniques for efficiently shadowing every byte of memory. Like their implementation, our shadow memory implementation also optimizes for common cases.

7.5 Call-path collection and data-centric attribution

We discuss related work in call-path collection in Section 7.5.1 and related work in data-centric attribution in Section 7.5.2.

7.5.1 Techniques for call-path collection

There are two principal techniques used to collect call-paths in an execution: unwinding the call stack and maintaining a shadow call stack.

Call stack unwinding on `x86` architectures walks procedure frames on the call stack using either compiler-recorded information (e.g., libunwind [161]) or results from binary analysis (e.g., HPCToolkit [2]). Stack unwinding does not require instrumentation and it does not maintain state information at each call and return. As a result, it adds no execution overhead, except when a stack trace is requested. This technique is well suited for coarse-grain execution monitoring, e.g., sampling-based performance tools and debuggers. Stack unwinding on each executed instruction, however, would frequently gather a slowly changing calling-context prefix at unacceptably high overhead.

Stack shadowing involves maintaining calling context as the execution unfolds. Instrumenting every function entry and exit, either at compile time or using binary rewriting, enables maintaining a shadow stack. Scalasca [75] and TAU [147], among other tools, employ a shadow stack. The advantage of stack shadowing is that the call-path is ready at every instant; and hence the call-path can be collected in a constant time. Stack shadowing is well suited for FEM tools. A disadvantage of stack shadowing is that the instrumentation of calls and returns adds overhead. Furthermore, it is not stateless, and it requires extra space to maintain the shadow stack.

Combining call stack unwinding and stack shadowing yields hybrid call-path collection techniques explored by Liu et al. [135] and Szebenyi et al. [220].

Dyninst [32] provides a stack walker API that can be queried on any instruction. Since DynInst uses unwinding for call-path collection, the overhead of doing so on each instruction makes it infeasible for FEM tools.

Valgrind [168] also provides a stack-unwinding framework for use by client tools that is unsuitable for fine-grained call-path collection. Callgrind, a Valgrind tool, maintains a shadow stack, which makes it possible to recover call-paths at a finer grain. However, instead of maintaining a CCT, Callgrind maintains a directory of call-paths at greater expense. Furthermore, the shadow stack maintained by Callgrind is not exported for use by other Valgrind tools.

Although at one time DynamoRio maintained a "software return stack" for improving branch prediction [30], it was judged unsuccessful; and a call-path abstraction was never made available as part of the tool's interface.

Whole Program Path (WPP) [126] is a technique to capture a program's inter-procedural control flow for the entire execution. Unlike call-path collection, which is a profiling mechanism, WPP is a trace collection mechanism. Since traces are more exhaustive than profiles, one can reconstruct call-paths from traces. Since traces can grow large, WPP performs an online compression of traces using the SEQUITUR hierarchical compression algorithm [169], which builds a context-free grammar for a string. WPP is useful for post-mortem analysis of execution for hot-path detection. WPP is not suitable for collecting contextual information for each interesting event such as a dead write collected throughout the execution since one

needs to maintain a prohibitively large number of unique markers that point into a WPP.

Efficient and compact calling context collection has attracted research interest in the recent times. Bond and McKinley [25] proposed an approach to maintain the calling context probabilistically. Instrumenting each call site and using a non-commutative hash function, allows them to maintain an integer value (32-bit or 64-bit) that uniquely represents the current calling context with minimum hash collisions Their overhead of instrumentation is under 3% on average for Java programs. D'Elia et al. [60] have developed an efficient algorithms for pruning large CCT's on the fly and maintaining a "Hot Calling Context Tree" (HCCT)—a subtree of the CCT that includes only hot nodes and their ancestors. They apply Cormode and Hadjieleftheriou's algorithm of finding frequent items in a data stream [56] to identify hot call-paths. HCCT technique is valuable for performance analysis tools; however, it may be limited in its use in correctness tools (e.g., data race detection), where call-paths leading to a bug may not appear on the hot paths.

7.5.2 Techniques for data-centric attribution

Performance tools such as MemProf [122], Memphis [151], and HPCTOOLKIT [134] employ data-centric attribution to diagnose memory-related bottlenecks. Memspy [150] monitors memory accesses using a cache simulator. Memspy attributes accesses to only heap variables. In many programs, static variables are also of interest. MACPO [193] uses LLVM [139] to instrument memory accesses and attribute accesses to variables. Though MACPO-generated code has comparatively low overhead (under 6×), it requires compilation with LLVM, and it is problematic to attribute costs in dynamically loaded libraries that are not compiled with LLVM. Unlike MACPO, CCTLib works with any `x86` compiler. Finally, ThreadSpotter [198] employs a "last write" method for data-centric attribution. It tracks store operations and uses them to identify variables at the source code level. This technique, which has an overhead of under 20%, leverages the fact that usually there is only one store (LHS value) in a source line.

Chapter 8

Conclusions and Future Work

I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right.

Albert Einstein

In this dissertation, we explored the implications of modern computer architectures on the application software performance. Rapid hardware advancements, explosion of parallelism, heterogeneity of processing cores, and deep memory hierarchies, make obtaining performance with modern architectures a daunting challenge. Partitioning of work, overheads of parallelism, choice of algorithms and data structures, design of abstractions, and characteristics of an underlying architecture have profound influence on the performance of an application.

We took a top-down approach to analyzing and improving the performance of an application in this dissertation. First, we explored the problem of systemic idleness arising primarily from improper work partitioning. Second, we explored the problem of synchronization overheads arising from barriers and locks. Third, we explored the problem of performance overheads arising from wasteful memory accesses. Throughout this dissertation, we followed the philosophy of attributing performance to its source code along with its calling context.

In the context of idleness analysis, this thesis demonstrated that the well-known blame shifting paradigm is an effective way to diagnose performance problems on accelerated architectures. Lightweight sampling-driven approach for profiling and tracing is useful for diagnosing not only software problems but also hardware problems. While we developed several strategies to work around limitations of accelerator hardware and software, we strongly believe that the accelerator vendors must provide better capabilities that a performance tool can leverage. The success of future accelerated software applications relies on the success of

performance tools for accelerated architectures.

Our blame-shifting technique identifies a "first-order" suspect. The true culprit, however, may be further removed. Blaming kernels instantaneously proved useful in our experience. However, future exascale applications might employ more asynchrony and need more sophisticated analysis to identify predecessors that delayed a GPU kernel charged using instantaneous blaming. Unifying our CPU-GPU blame shifting with the blame-shifting techniques for work-stealing and lock contention is yet another avenue for our future work.

In the context of the overhead of barrier synchronization, we developed a context-sensitive, dynamic program analyses and transformation schemes to elide unnecessary barriers in Partitioned Global Address Space programs. While we expected some redundant barriers in NWChem, the magnitude (60% redundant barriers) was surprising to everybody, including NWChem experts. Our runtime technique for barrier elision saves about 14% execution time, which is a respectable performance gain for large scientific executions that consume millions of machine hours. Since scientific codes are evolving towards multi-physics multi-scale simulations and increasingly using PGAS or one-sided communication, it is likely that redundant synchronization will become more prevalent.

While this dissertation explored redundant barriers in distributed memory parallel programs, other venues of redundancy include barriers in shared-memory programs. We also foresee eliminating redundant memory fences and converting blocking communication into non-blocking communication as other extensions to our barrier elision work.

In the context of locks, we developed a NUMA-aware lock that delivers top performance on machines with deep and distributed memory hierarchies. Our work is the first one to develop analytical models for throughput and fairness for queuing locks. Analytically modeling hierarchical queuing locks was particularly challenging. Our analytical models provide insight into properties of hierarchical lock designs. Experiments confirm the accuracy of our models. On systems with more than two levels of NUMA hierarchy, hierarchical MCS (HMCS) locks deliver high throughput with significantly less unfairness than the previously designed two-level locks. Differences in access latencies between NUMA domains at different levels of a hierarchy determine what levels of the hierarchy are worth exploiting. Given a measure of the passing time at each level of a hierarchy, our models show how to pick cohorting thresholds

that deliver a desired fraction of the maximum throughput.

We realized that a fixed-depth HMCS lock could hurt the performance of uncontended lock acquisition, especially when the hierarchy is deep. We addressed this problem with a combination of a fast-path and hysteresis techniques that make our hierarchical lock usually match the performance of an ideal hierarchical queuing lock for any contention level—low, medium, or high. We also explored the hardware transactional memory facility available in IBM POWER8 architecture and demonstrated the necessity of sophisticated locality-aware software mutual-exclusion schemes.

Our current implementation of the adaptive HMCS lock (AHMCS) uses hysteresis to move one level at a time in the locking hierarchy. We already have sufficient information to skip multiple levels of a hierarchy to improve our design. The AHMCS lock is a competitive algorithm that switches between various algorithms. We believe exploring the complexity bounds of the AHMCS lock, analogous to proving the bounds of an adaptive scheduling algorithm [4, 5, 219], will strengthen our design.

The NUMA-aware HMCS lock discussed in this dissertation does not allow a thread to abort an acquisition if the wait is too long. It is desirable to add timeout capability to locks analogous to [63, 205]. One can enhance the HMCS lock with the timeout capability. Besides providing the timeout capability, proving the correctness of such protocols is an important challenge.

While our experiments for locks to date were conducted on shared memory platforms, our work was motivated by distributed-memory systems that use Remote Direct Memory Access (RDMA) primitives to implement system-wide queueing locks. The latency of RDMA operations between nodes is much higher than the latency of loads and stores within nodes. We believe that HMCS locks are the best way to achieve high throughput on such systems. Implementing HMCS locks for such systems remains future work.

In the context of wasteful memory accesses, we developed a tool to pinpoint and quantify dead writes in an execution. We showed that several commonly used programs have surprisingly large fractions of dead writes. Significant amount of dead writes in flagship codes such as `gcc` and `NWChem` was surprising even for their expert developers.

The pervasiveness of dead writes suggests a new opportunity for performance tuning. We

recommend investigating and eliminating avoidable memory operations as an important step in performance tuning to identify opportunities for code restructuring. While fine-grained performance monitoring may introduce a high runtime overhead, executing programs with small representative workloads will help uncover regions of inefficiencies quickly.

One can develop several other fine-grained monitoring tools that can identify wasteful resource consumption. It is also conceivable to reduce a fine-grain monitoring tool's overhead by employing a combination of sampling and static binary analysis.

Finally, recognizing the importance of contextual performance attribution, this dissertation attributed performance metrics to source code and data objects, in context. We enhanced HPCTOOLKIT's calling context performance attribution to GPU tasks. We developed a mechanism to pinpoint redundant barriers in their context, which also provided an avenue to optimize redundant barriers in their calling context. We developed DEADSPY to pinpoint dead writes in their calling contexts.

Fine-grained execution profiling and attributing such profiles to calling contexts was perceived to be infeasible prior to our work in this dissertation. In this dissertation, we developed an open-source library for fine-grained performance attribution to calling context and data objects in the Pin binary analysis framework. This work enables several fine-grained instruction-level execution-monitoring tools used in performance analysis, software correctness, security, and other domains, to deliver detailed diagnostic feedback. Compared to state-of-the-art open-source Pin tools for call-path collection, our library is richer in information, more accurate even for programs with complex control flow and does so with lower overhead. With our library, one can collect call-paths and attribute memory accesses to data objects on every executed machine instruction even for reasonably long-running programs. Our library has modest runtime overhead and scales well when used on multithreaded codes.

Throughout this dissertation, we followed the philosophy of building sound principles of performance measurement and attribution. Once the causes of performance problems were identified, in each application we investigated, we addressed the performance problems on a case-by-case basis. Some solutions were tailored for an application (e.g., LULESH), whereas some others were generic algorithms (e.g., the HMCS lock).

The success of software applications depends on their performance. It would not be an

understatement to say that for computationally intensive applications on modern parallel systems, performance is as important as correctness.

Appendix A

Implementation of FP-AHMCS locks

In this appendix, we provide the implementation details of the adaptive HMCS locks. Section A.1 and Section A.2 describe the implementation details of the FP-EH-AHMCS (fast-path augmented eager hysteresis HMCS) and FP-LH-AHMCS (fast-path augmented lazy hysteresis HMCS) locks, respectively.

A.1 FP-EH-AHMCS lock

As discussed previously in Section 4.9, an AHMCS lock is a wrapper around the HMCS lock. The AHMCS, however, modifies the HMCS lock to return a value indicating the tree depth where the lock passing happened. The acquire protocol returns the depth where the thread noticed a predecessor. The release protocol returns the depth where the thread noticed a successor. Listing A.1 shows this modification. Lines marked with the "■" symbol indicate the deviations from the HMCS lock previously shown in Listing 4.1.

The core FP-EH-AHMCS algorithm that monitors and adapts to contention is shown in Listing A.2. During lock initialization, each thread obtains an `FP-EH-AHMCSLock` object. Each `FP-EH-AHMCSLock` object has the following thread-local fields:

1. `I`: is a `QNode` used to enqueue with the HMCS protocol.
2. `leafNode`: is the lowest-level `HNode` in the HMCS tree where the thread begins its enqueue process. This value will be different for threads of different domains.
3. `curNode`: is the current `HNode` in the HMCS tree, where the thread will be targeting its enqueue process, initially `leafNode`.
4. `childNode`: is the `HNode` immediately below the `curNode` in the path from `curNode` to `leafNode` in the HMCS tree, initially NULL.

5. `rootNodeNode`: is the root-level `HNode`.

6. `tookFastPath`: is a boolean indicating if the protocol took the fast-path.

7. `realStartDepth`: is the depth where the enqueue process actually started for the current round.

8. `realHNodeAtEnqueue`: is the `HNode` where the enqueue process actually started for the current round.

9. `curDepth`: is the current depth that the thread targets its acquisition protocol to start in the HMCS tree, initially equal to the `MaxDepth`.

10. `depthWaited`: is an integer that records the depth where the acquisition protocol found a predecessor.

11. `depthPassed`: is an integer that records the depth where the release protocol found a successor.

The `FP-EH-AHMCSLock` class has the following static fields:

1. `AHMCSAcquireFnTable`: A table of function pointers to each level of the HMCS lock acquire functions.

2. `AHMCSReleaseFnTable`: A table of function pointers to each level of the HMCS lock release functions.

These pointers are needed so that the `Acquire` and `Release` functions of a `FP-EH-AHMCSLock` object can dynamically switch from one level of acquisition to the other. We explain the key details of the algorithm shown in Listing A.2 below.

In the acquire protocol, if both the current level lock (line 34) and the root level lock (line 36) are uncontended, then the fast-path is taken. The fast-path invokes the the HMCS$\langle 1 \rangle$'s acquire routine (line 39). If there is child level and the current level has sufficient contention (line 45), we eagerly decide to enqueue at the child level. An exception to this norm is if the current tail pointer of the lock is same as the successor of the current thread

in the previous iteration; in such cases, the contention may not be high enough to justify the detour through the child level.

In the default case, we enqueue at the current level (lines 53-54). We memorize the "real" depth where we started the acquisition (`realStartDepth`) and the `HNode` at the point of acquisition (`realHNodeAtEnqueue`). We invoke the base HMCS acquire protocol via a pre-populated function pointer table and record the depth where the acquisition found a predecessor in the variable `depthWaited` (line 58).

In the release protocol, if the `tookFastPath` flag is set (line 62) we invoke the HMCS$\langle 1 \rangle$'s release protocol and unset `tookFastPath`. Otherwise, we invoke the base HMCS release protocol via a pre-populated function pointer table indexed at `realStartDepth` and use the `HNode` as `realHNodeAtEnqueue` (line 71). This call allows us to obtain the depth where the release protocol first found a successor, which we record in the variable `depthPassed`.

The key logic to adjust to contention appears in lines 76-96. If either the acquire phase found a predecessor or the release phase found a successor closer to leaves (line 76), we increment the depth by one and adjust the `curNode` and `childNode` accordingly. Otherwise, if the acquire phase found a predecessor and the release phase found a successor closer to the root (line 91), we decrement the depth (`curDepth`) by one and adjust the `curNode` and `childNode` accordingly.

```
1  enum {COHORT_START=0x1, ACQUIRE_PARENT=UINT64_MAX-1, WAIT=UINT64_MAX};
2  enum {UNLOCKED=0x0, LOCKED=0x1};
3
4  template<int depth> struct HMCSLock {
5    inline int AcquireReal(HNode *L, QNode *I) {
6      // Prepare the node for use.
7      I->status = WAIT; I->next = NULL;
8      ------------------- release fence -------------------
9      QNode * pred = (QNode *) SWAP(&(L->lock), I);
10     if (pred) {
11       pred->next = I;
12       uint64_t curStatus;
13       while( (curStatus=I->status) == WAIT); // spin
14       if(curStatus < ACQUIRE_PARENT)
15 ■       // Acquired lock from a predecessor at this depth
16 ■       return depth;
17     }
18     I->status = COHORT_START;
19 ■   // This level was acquired without contention, return the level where we first found a predecessor
20 ■   return HMCSLock<depth-1>::AcquireReal(L->parent, &(L->node));
21   }
22
23   inline int Acquire(HNode *L, QNode *I) {
24 ■   int whereAcquired = AcquireReal(L, I);
25     ------------------- acquire fence -------------------
26 ■   return whereAcquired;
27   }
28
29   inline int ReleaseReal(HNode *L, QNode *I) {
30     uint64_t curCount = I->status;
31     // Lower level releases
32     if (curCount == L->GetThreshold()) {
33       // reached threshold, release to next level
34       HMCSLock<depth-1>::ReleaseReal(L->parent, &(L->node));
35       ------------------- release fence -------------------
36       // Ask successor to acquire next level lock
37       ReleaseHelper(L, I, ACQUIRE_PARENT);
38 ■     // if we passed many times, we can simply assume this one would have found a successor as well
39 ■     return depth;
40     }
41     // Not reached threshold
42     QNode * succ = I->next;
43     if (succ) {  // Pass within cohorts
44       succ->status = curCount + 1;
45 ■     // We passed to a successor at this depth
46 ■     return depth;
47     }
48     // else: No known successor, release to parent
49 ■   int whereReleased = HMCSLock<depth-1>::ReleaseReal(L->parent, &(L->node));
50     ------------------- release fence -------------------
51     // Ask successor to acquire next level lock, we can not make use of passing
52     ReleaseHelper(L,I, ACQUIRE_PARENT);
53 ■   // Did not observe the possibility of passing at this depth
54 ■   return whereReleased;
55   }
56
57   inline int Release(HNode *L, QNode *I) {
58     ------------------- release fence -------------------
59 ■   return ReleaseReal(L, I);
60   }
61 };
62
63   inline void ReleaseHelper(HNode *L, QNode *I, uint64_t val){
64     QNode * succ = I->next;
65     if (succ) {
66       succ->status = val; // pass lock
67       return;
68     } else {
69       if (CAS(&(L->lock), I, NULL))
70         return;
71       while( (succ=I->next) == NULL);  // spin
72       succ->status = val; // pass lock
73       return;
74     }
75   }
76
77 template <> struct HMCSLock<1> {
78   inline int AcquireReal(HNode *L, QNode *I) {
79     // Prepare the node for use.
80     I->status = LOCKED; I->next = NULL;
81     QNode * pred = (QNode *) SWAP(&(L->lock), I);
82     if (!pred) {
83       I->status = UNLOCKED;
84 ■     // Acquired at depth 1
85 ■     return 1;
86     } else {
87       pred->next = I;
88       while(I->status == LOCKED); // spin
89 ■     // Acquired at depth 1
90 ■     return 1;
91     }
92   }
93
94   inline int Acquire(HNode *L, QNode *I) {
95 ■   int whereAcquired = = AcquireReal(L, I);
96     ------------------- acquire fence -------------------
97 ■   return whereAcquired;
98   }
99
100  inline int ReleaseReal(HNode *L, QNode *I) {
101 ■   return ReleaseHelper(L, I, UNLOCKED);
102  }
103
104  inline int Release(HNode *L, QNode *I) {
105    ------------------- release fence -------------------
106 ■   return ReleaseReal(L, I);
107  }
108 };
```

Listing A.1: HMCS lock adapted to return the depth at which the lock was passed.

```
template<int MaxDepth>
struct FP-EH-AHMCSLock{
  QNode I; // The node this thread will use for enqueueing
  // Needed for taking the fast-path
  HNode * rootNode; // Root lock of the HMCS tree
  bool tookFastPath; // Did we take fast-path?
  // Needed for adaption
  HNode * leafNode; // The leaf HNode in the HMCS tree
  HNode * curNode; // The current HNode in the HMCS tree, where the enqueue begins
  HNode * childNode; // The HNode just one below the curNode (for checking the tail pointer and contentionCounter)
  HNode * realHNodeAtEnqueue; // The real HNode where we enqueued ultimately (either curNode or childNode)
  uint8_t curDepth; // Current depth where the enqueue beings
  uint8_t realStartDepth; //  // The real depth where we actually begun acquisition
  uint8_t depthWaited; // depth where the acquisition protocol found a predecessor
  uint8_t depthPassed; // depth where the release protocol found a successor
  typedef int (*HMCSFnPtr)(HNode *, QNode *);
  // A function pointer table to different levels of the HMCS acquire protocols
  // Initialized with template metaprogramming in the actual code.
  static HMCSFnPtr AHMCSAcquireFnTable[MaxDepth]={0, HMCSLock<1>::Acquire, ..., HMCSLock<MaxDepth>::Acquire};
  // A function pointer table to different levels of the HMCS release protocols
  // Initialized with template metaprogramming in the actual code.
  static HMCSFnPtr AHMCSReleaseFnTable[MaxDepth]={0, HMCSLock<1>::Release, ..., HMCSLock<MaxDepth>::Release};

  // Constructor
  FP-EH-AHMCSLock(HNode * leaf) :
    curNode(leaf), leafNode(leaf), childNode(NULL),
    curDepth(MaxDepth),
    tookFastPath(false){
      // Initialize the root node
    }
  inline void Acquire(){
    QNode * curTail;
    // Is current level contended?
    if( (curTail=curNode->lock) == NULL){
      // Is root level contended?
      if(rootNode->lock == NULL){Slow-path
        // Root is uncontended, take fast-path
        tookFastPath = true;
        HMCSLock<1>::Acquire(rootNode, &I);
        return;
      }
      // Root is contended, take slow-path
      realStartDepth = curDepth;
      realHNodeAtEnqueue = curNode;
   } else if(childNode && (I->next && I->next != curTail)){
      // current level is sufficiently contended
      // Eagerly enqueue at a level below
      realStartDepth = curDepth + 1;
      realHNodeAtEnqueue = childNode;
    } else {
      // Either no child level or
      // the current level is not sufficiently contended
      realStartDepth = curDepth;
      realHNodeAtEnqueue = curNode;
    }
    // Slow-path
    // Call the HMCS lock acquire through the function pointer for the current depth of acquire (curDepth)
    depthWaited = FP-EH-AHMCSLock::AHMCSAcquireFnTable[realStartDepth](realHNodeAtEnqueue, &I);
  }

  inline void Release(){
    if(tookFastPath) {
      // Fast-path
      HMCSLock<1>::Release(rootNode, &I);
      // Unset the fast-path for next round
      tookFastPath = false;
      return;
    }
    // Fast-path
    // Call the HMCS lock release through the function pointer for the current depth of acquire (curDepth)
     depthPassed = FP-EH-AHMCSLock::AHMCSReleaseFnTable[curDepth](curNode, &I) ;

    //Key logic to adjust to contention
    // If we acquired and released closer to the leaf in the tree, compare to
    // the currently noted depth, we recede a level closer to the leaf
    if(childNode && ( (depthWaited > curDepth) || (depthlFirstPassed > curDepth)) ){
      curDepth++;
      curNode = childNode;
      if(curDepth == maxLevels) {
        childNode = NULL;
        return;
      }
      // Find the new childNode
      HNode * temp;
      for(temp = leafNode; temp->parent != childNode; temp = temp->parent);
      childNode = temp;
      return;
    }
    // If we acquired and released closer to the root in the tree, compare to
    // the currently noted depth, we rise a level closer to the root
    if ((curDepth != 1) && ( (depthWaited <  curDepth) && (depthlFirstPassed < curDepth))){
      curDepth--;
      childNode = curNode;
      curNode = curNode->parent;
      return;
    }
  }
};
```

Listing A.2: Eager Adaptive HMCS lock algorithm for adjusting to contention.

A.2 FP-LH-AHMCS lock

The FP-LH-AHMCS lock also uses the same modified HMCS lock shown in Listing A.1. The FP-LH-AHMCS algorithm introduces a `contentionCounter` field in each `HNode`. The core algorithm that monitors and adapts to contention is shown in Listing A.3. During lock initialization, each thread obtains a `FP-LH-AHMCSLock` object. Each `FP-LH-AHMCSLock` object has the same fields as the `FP-EH-AHMCSLock`. In addition, the `FP-LH-AHMCSLock` has the following fields.

1. `childContentionOnAcquire` and `childContentionOnRelease`: are booleans that record if there was contention within the subdomain before starting acquisition and after ending the release, respectively.

2. `lastSeenCounter`: is an integer that remembers the value of the `contentionCounter` seen at the `childNode` just before the acquisition process.

3. `hysteresis`: is an integer between `MOVE_UP` and `MOVE_DOWN`. Each time an uncontended acquisition and release is performed, the `hysteresis` is decremented. Each time a contended acquisition and release is performed, the `hysteresis` is incremented. Otherwise, `hysteresis` is reset to `STAY_PUT`, a value s.t., `MOVE_UP` $<$ `STAY_PUT` $<$ `MOVE_DOWN`.

We explain the key details of the algorithm shown in Listing A.3 below. The fast-path mechanism is exactly same as previously shown in the FP-EH-AHMCS lock, hence we do not show its details here (the fast-path logic would appear before lines 58 and 65). The `Acquire` routine calls `OnAcquire` on line 58, which records whether there was contention in the subdomain of the thread on entry. Then, the routine calls the base HMCS lock's acquire through the function pointer for the appropriate depth where the acquisition is supposed to begin.

The `Release` routine calls the base HMCS lock's release through the function pointer for the appropriate depth where the release is supposed to begin. Then, the routine calls the function `OnRelease` on line 58, which records whether there was contention in the subdomain of the thread on exit.

Lines 68-101 contain the key logic to adjust to contention. If `childContentionOnAcquire` and `childContentionOnRelease` are set, it means, there is enough contention in the sub-domain that it may be possible to exploit locality by enqueuing at a level closer to leaves in the HMCS tree (line 68). If the situation has been observed for `MOVE_DOWN` number of times, then the algorithm adjusts its fields so that next acquisition begins one level below (line 70). Since the new `childNode` is not directly available, the algorithm climbs the tree from a leaf to the current node looking for the new child node (lines 76-80). Once the level is changed, the `hysteresis` is reset (line 81). If the `hysteresis` has not reached the threshold, then it is decremented (line 83). Observe that, if on one round the indication is low contention and the next round the indication is high contention, then the hysteresis resists spontaneous changes.

A low contention is recognized via the same strategy as in the `FP-EH-AHMCSLock` lock (line 87). If the low contention is observed for `MOVE_UP` number of times, then the algorithm adjusts its fields so that the next acquisition begins one level closer to the root (line 89). Once the level is changed, the `hysteresis` is reset. If the `hysteresis` has not reached the threshold, then it is incremented (line 95).

If the contention neither increases not decreases, we reset the hysterisis (line 100).

```
template<int MaxDepth>
struct FP-LH-AHMCSLock{
  QNode I; // The node this thread will use for enqueueing
  // Needed for taking the fast-path
  HNode * rootNode; // Root lock of the HMCS tree
  bool tookFastPath; // Did we take fast-path?
  HNode * leafNode; // The leaf HNode in the HMCS tree
  HNode * curNode; // The current HNode in the HMCS tree, where the enqueue begins
  HNode * childNode; // The HNode just one below the curNode (for checking the tail pointer and contentionCounter)
  uint8_t curDepth; // Current depth where the enqueue beings
  bool childContentionOnAcquire; // Was there contention in the subdomain during entry (acquire)
  bool childContentionOnRelease; // Was there contention in the subdomain during exit (release)
  uint8_t depthWaited; // depth where the acquisition protocol found a predecessor
  uint8_t depthPassed; // depth where the release protocol found a successor
  bool hasCounterChanged; // has the counter value changed from acquire to release
  int8_t hysteresis; // How many times a contention information is observed
  uint64_t lastSeenCounter; // The contentionCounter at childNode on entry
  typedef int (*HMCSFnPtr)(HNode *, QNode *);
  // A function pointer table to different levels of the HMCS acquire protocols
  // Initialized with template metaprogramming in the actual code.
  static HMCSFnPtr AHMCSAcquireFnTable[MaxDepth]={0, HMCSLock<1>::Acquire, ..., HMCSLock<MaxDepth>::Acquire};
  // A function pointer table to different levels of the HMCS release protocols
  // Initialized with template metaprogramming in the actual code.
  static HMCSFnPtr AHMCSReleaseFnTable[MaxDepth]={0, HMCSLock<1>::Release, ..., HMCSLock<MaxDepth>::Release};

  // Constructor
  FP-LH-AHMCSLock(HNode * leaf) :
    hysteresis(STAY_PUT),   curDepth(MaxDepth),
    curNode(leaf), leafNode(leaf), childNode(NULL),
    childContentionOnAcquire(false), childContentionOnRelease(false) {
     // Setup
    }

  inline void OnAcquire(){
    if (childNode) {
      // Remember the contentionCounter seen at the entry
      lastSeenCounter =  ADD_AND_FETCH(&childNode->contentionCounter, 1);
      // If the tail pointer of the childNode is non-null, then there was contention during entry
      childContentionOnAcquire = childNode->lock !=  NULL;
    } else {
      childContentionOnAcquire = false;
    }
  }

  inline void OnRelease(){
    if (childNode) {
      // Is the tail pointer of the childNode is non-null?
      childContentionOnRelease = childNode->lock != NULL;
      // Has the contentionCounter changed?
      hasCounterChanged = lastSeenCounter != childNode->contentionCounter;
    } else {
      childContentionOnRelease = false;
      hasCounterChanged = false;
    }
  }

  inline void Acquire(){
    OnAcquire();  // Record contention seen on acquire
    // Call the HMCS lock acquire through the function pointer for the current depth of acquire (curDepth)
    depthFirstAcquired = FP-LH-AHMCSLock::AHMCSAcquireFnTable[curDepth](curNode, &I);
  }

  inline void Release(){
    // Call the HMCS lock release through the function pointer for the current depth of acquire (curDepth)
    depthFirstReleased = FP-LH-AHMCSLock::AHMCSReleaseFnTable[curDepth](curNode, &I);
    OnRelease(); // Record contention seen on release
    //Key logic to adjust to contention
    if(childNode && ( (childContentionOnAcquire && childContentionOnRelease) || hasCounterChanged)){
      // Have contention in my subdomain
      if(hysteresis == MOVE_DOWN) { // Reached the threshold of contention
        curDepth++; // Move one level down
        curNode = childNode;
        if(curDepth == MaxDepth) {
          childNode = NULL;
        } else {
          // Find the new childNode by walking up from the leafNode.
          HNode * temp;
          for(temp = leafNode; temp->parent != childNode; temp = temp->parent);
          childNode = temp;
        }
        hysteresis = STAY_PUT; // Reset the hysteresis
      } else {  // Not reached the threshold
        hysteresis--; // Decrement the hysteresis
      }
      return;
    }
    if ((curDepth != 1) && ((depthWaited < curDepth) && (depthPassed < curDepth))){
      // No contention in my subdomain
      if(hysteresis == MOVE_UP) { // Reached the threshold  of no contention in my subdomain
        curDepth--; // Move one level up in the HMCS tree
        childNode = curNode;
        curNode = curNode->parent;
        hysteresis = STAY_PUT;
      } else { // Not reached the threshold
        hysteresis++; // Increment the hysteresis
      }
      return;
    }
    // Contention is just right enough to stay at the same level, hence reset the hysteresis
    hysteresis = STAY_PUT;
  }
};
```

Listing A.3: Lazy adaptive HMCS lock algorithm for adjusting to contention.

Bibliography

[1] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato. Power and Performance Analysis of GPU-accelerated Systems. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, April 2010.

[3] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel Analysis of X10 Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM.

[4] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive Scheduling with Parallelism Feedback. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 100–109, New York, NY, USA, 2006. ACM.

[5] K. Agrawal, Y. He, and C. E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 112–120, New York, NY, USA, 2007. ACM.

[6] A. M. Aji, L. S. Panwar, F. Ji, M. Chabbi, K. Murthy, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey, X. Ma, and R. Thakur. On the Efficacy of GPU-integrated MPI for Scientific Applications. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 191–202, New York, NY, USA, 2013. ACM.

[7] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI Collective Communication

on BlueGene/L Systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[8] AMD Corp. AMD64 Architecture Programmer's Manual, Volume 2: System Programming. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`, 2012.

[9] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, May 1997. ACM.

[10] J. H. Anderson and Y.-J. Kim. A New Fast-path Mechanism for Mutual Exclusion. *Distributed Computing*, 14(1):17–29, Jan. 2001.

[11] Applied Numerical Algorithms Group, Lawrence Berkeley National Laboratory. Chombo Website. `https://seesar.lbl.gov/anag/chombo`.

[12] R. G. Archambault. Interprocedural dead store elimination. `http://www.patentlens.net/patentlens/patent/US_7100156/en/`, 08 2006. US 7100156.

[13] E. M. Asimakopoulou. cptnHook. `https://github.com/emyrto/cptnHook`, 2015.

[14] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu. Efficient Performance Evaluation of Memory Hierarchy for Highly Multithreaded Graphics Processors. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 23–34, New York, NY, USA, 2012. ACM.

[15] P. Balaji. MPICH 3.0.4 Released. `https://www.mpich.org/2013/04/25/mpich-3-0-4-released/`.

[16] R. Balasubramonian, N. Jouppi, and N. Muralimanohar. *Multi-Core Cache Hierarchies: Recent Advances*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2011.

[17] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, July 1994.

[18] L. A. Belady. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal*, 5(2):78–101, June 1966.

[19] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[20] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[21] R. Bodík and R. Gupta. Partial Dead Code Elimination Using Slicing Transformations. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 159–170, New York, NY, USA, 1997. ACM.

[22] D. Bohme, M. Geimer, F. Wolf, and L. Arnold. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 90–100, Washington, DC, USA, 2010. IEEE Computer Society.

[23] D. Bohme, F. Wolf, B. de Supinski, M. Schulz, and M. Geimer. Scalable Critical-Path Based Performance Analysis. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1330–1340, May 2012.

[24] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, Winter 2007.

[25] M. D. Bond and K. S. McKinley. Probabilistic Calling Context. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 97–112, New York, NY, USA, 2007. ACM.

[26] M. D. Bond, V. Srivastava, K. S. McKinley, and V. Shmatikov. Efficient, Context-sensitive Detection of Real-world Semantic Attacks. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 1:1–1:10, New York, NY, USA, 2010. ACM.

[27] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable Locks are Dangerous. In *Proceedings of Linux Symposium*, 2012.

[28] M. Broberg, L. Lundberg, and H. Grahn. Performance Optimization Using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors. *Journal of Parallel and Distributed Computing*, 61(1):115–136, Jan. 2001.

[29] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing molecular dynamics on hybrid high performance computers - short range forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[30] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Dept. of EECS, MIT, September 2004.

[31] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[32] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[33] P. A. Buhr, D. Dice, and W. H. Hesselink. High-performance N-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3):651–701, 2015.

[34] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 199–210, New York, NY, USA, 2002. ACM.

[35] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. ACM.

[36] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM.

[37] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[38] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[39] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 215–226, New York, NY, USA, 2015. ACM.

[40] M. Chabbi, W. Lavrijsen, W. de Jong, K. Sen, J. Mellor-Crummey, and C. Iancu. Barrier Elision for Production Parallel Programs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 109–119, New York, NY, USA, 2015. ACM.

[41] M. Chabbi, X. Liu, and J. Mellor-Crummey. Call Paths for Pin Tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM.

[42] M. Chabbi and J. Mellor-Crummey. DeadSpy: A Tool to Pinpoint Program Inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 124–134, New York, NY, USA, 2012. ACM.

[43] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey. Effective Sampling-driven Performance Tools for GPU-accelerated Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 43:1–43:12, New York, NY, USA, 2013. ACM.

[44] G. J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[45] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[46] S. Chamberlain. Using ld, The GNU linker. `ftp://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html`, 1994.

[47] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 264–276, Washington, DC, USA, 2006. IEEE Computer Society.

[48] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[49] C. Charlie and B. Emery. Coz: Finding Code that Counts with Causal Profiling. `https://web.cs.umass.edu/publication/docs/2015/UM-CS-2015-008.pdf`, March 2015.

[50] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating Asynchronous Task Parallelism with MPI. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 712–725, May 2013.

[51] G. Chen and P. Stenstrom. Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 71:1–71:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[52] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 357–368, Washington, DC, USA, 2005. IEEE Computer Society.

[53] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability Analysis of SPMD Codes Using Expectations. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 13–22, New York, NY, USA, 2007. ACM.

[54] ComEx: Communications Runtime for Exascale. `http://hpc.pnl.gov/comex/`.

[55] K. Cooper, J. Eckhardt, and K. Kennedy. Redundancy Elimination Revisited. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 12–21, New York, NY, USA, 2008. ACM.

[56] G. Cormode and M. Hadjieleftheriou. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, Aug. 2008.

[57] Cray. Using the GNI and DMAPP APIs, version S-2446-3103. `http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf`, 2011.

[58] Dave Goodwin, Nvidia Corp. Personal communication.

[59] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, New York, NY, USA, 2013. ACM.

[60] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining Hot Calling Contexts in Small Space. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 516–527, New York, NY, USA, 2011. ACM.

[61] D. Dice, D. Hendler, and I. Mirsky. Lightweight Contention Management for Efficient Compare-and-swap Operations. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 595–606, Berlin, Heidelberg, 2013. Springer-Verlag.

[62] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[63] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[64] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Trans. Parallel Comput.*, 1(2):13:1–13:42, Feb. 2015.

[65] P. C. Diniz and M. C. Rinard. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-based Programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, Mar. 1998.

[66] Dormando et al. Memcached. `http://memcached.org/`, 2015.

[67] Y. Dotsenko. *Expressiveness, Programmability and Portable High Performance of Global Address Space Languages*. PhD thesis, Dept. of CS, Rice University, January 2007.

[68] T. Duff. Tom Duff on Duff's Device. `http://www.lysator.liu.se/c/duffs-device.html`.

[69] DynInst API Developers. DynInst API Mailing List Archives. `https://www-auth.cs.wisc.edu/lists/dyninst-api/2013/msg00135.shtml`, 2013.

[70] J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. Stone, and J. Phillips. Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters. In *Green Computing Conference, 2010 International*, pages 317–324, Aug 2010.

[71] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 103:1–103:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[72] P. Fatourou and N. D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[73] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[74] GCC Team. The GNU Compiler Collection. `http://gcc.gnu.org/`.

[75] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurrency Computation: Practice and Experience*, 22(6):702–719, Apr. 2010.

[76] Google Inc. Sparsehash, version 2.0.2. `https://code.google.com/p/sparsehash/`, February 2012.

[77] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 258–264, New York, NY, USA, 2005. ACM.

[78] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[79] R. Gupta, D. A. Berson, and J. Z. Fang. Path Profile Guided Partial Dead Code Elimination Using Predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, pages 102–, Washington, DC, USA, 1997. IEEE Computer Society.

[80] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun. On-the-fly Detection of Data Races in OpenMP Programs. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2012, pages 1–10, New York, NY, USA, 2012. ACM.

[81] D. Hackenberg, G. Juckeland, and H. Brunst. Performance analysis of multi-level parallelism: inter-node, intra-node and hardware accelerators. *Concurrency and Computation: Practice and Experience*, 24(1):62–72, 2012.

[82] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[83] L. C. Harris and B. P. Miller. Practical Analysis of Stripped Binary Code. *ACM SIGARCH Computer Architecture News — Special issue on the 2005 workshop on binary instrumentation and application*, 33(5):63–68, Dec. 2005.

[84] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[85] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture Newss*, 34(4):1–17, Sept. 2006.

[86] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[87] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[88] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software (TOMS) - Special issue on the Advanced Computational Software (ACTS) Collection*, 31(3):397–423, Sept. 2005.

[89] P. Hicks, M. Walnock, and R. M. Owens. Analysis of Power Consumption in Memory Hierarchies. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 239–242, New York, NY, USA, 1997. ACM.

[90] T. Hoefler. Evaluation of Publicly Available Barrier-Algorithms and Improvement of the Barrier-Operation for large-scale Cluster-Systems with special Attention on InfiniBand Networks, Apr. 2005.

[91] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[92] S. Huang, S. Xiao, and W. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[93] IBM. Power ISA Version 2.06 Revision B. `https://www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf`, 2010.

[94] W. N. S. III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory, 2004.

[95] InfiniBand Trade Association. InfiniBand® Architecture Volume 1 and Volume 2. `http://www.infinibandta.org/content/pages.php?pg=technology_public_specification`, 2013.

[96] Intel Corp. Intel Compilers. `http://software.intel.com/en-us/articles/intel-compilers/`.

[97] Intel Corp. An Introduction to the Intel® QuickPath Interconnect. `http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html`, 2009.

[98] Intel Corp. Hotspots Analysis. https://software.intel.com/sites/products/documentation/doclib/iss/2013/amplifier/lin/ug_docs/GUID-4D83FDF6-AD8C-478C-BE00-B7527EA3A897.htm, 2013.

[99] Intel Corp. Intel Pin. http://software.intel.com/en-us/articles/pintool-downloads, 2013.

[100] Intel Corp. Intel® Xeon Phi Coprocessor–the Architecture. https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, 2013.

[101] Intel Corp. 4th Generation Intel® Core Processor. https://software.intel.com/en-us/haswell, 2015.

[102] B. L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[103] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[104] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[105] T. E. Jeremiassen and S. J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Programs. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '94, pages 171–180, Amsterdam, The Netherlands, 1994. North-Holland Publishing Co.

[106] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. NAS Technical Report NAS-99-011, NASA Advanced Supercomputing Division, 1999.

[107] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP Scalability Using Speculative Lock Inheritance. *Proceedings of the VLDB Endowment*, 2(1):479–489, Aug. 2009.

[108] T. Johnson and K. Harathi. A Simple Correctness Proof of the MCS Contention-free Lock. *Information Processing Letters*, 48(5):215–220, Dec. 1993.

[109] A. Kamil and K. Yelick. Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 185–199, Berlin, Heidelberg, 2006. Springer-Verlag.

[110] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory, December 2012.

[111] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[112] I. Karlin, J. Keasler, and R. Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[113] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *Micro, IEEE*, 31(5):7–17, Sept 2011.

[114] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 335–346, New York, NY, USA, 2011. ACM.

[115] C. Kim, D. Burger, and S. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *Micro, IEEE*, 23(6):99–107, Nov 2003.

[116] A. Kleen. Lock elision in the GNU C library. `https://lwn.net/Articles/534758/`, 2013.

[117] J. Knoop, O. Rüthing, and B. Steffen. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM.

[118] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric Multi-level Blocking. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 346–357, New York, NY, USA, 1997. ACM.

[119] A. Kogan and E. Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[120] A. Kogan and E. Petrank. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 141–150, New York, NY, USA, 2012. ACM.

[121] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 763–773, May 2012.

[122] R. Lachaize, B. Lepers, and V. Quéma. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[123] P.-W. Lai, K. Stock, S. Rajbhandari, S. Krishnamoorthy, and P. Sadayappan. A Framework for Load Balancing of Tensor Contraction Expressions via Dynamic Task Partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 13:1–13:10, New York, NY, USA, 2013. ACM.

[124] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, Jan. 1987.

[125] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[126] J. R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM.

[127] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, Jan 2015.

[128] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12):1352–1361, Dec. 2001.

[129] J. Lee, H. Kim, and R. Vuduc. When Prefetching Works, When It Doesn'T, and Why. *ACM Transactions on Architecture and Code Optimization*, 9(1):2:1–2:29, Mar. 2012.

[130] C. E. Leiserson. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[131] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[132] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-aware Monitored Execution. In *Proceedings of Network and Distributed System Security Symposium*, 2008.

[133] X. Liu and J. Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 171–180, Washington, DC, USA, 2011. IEEE Computer Society.

[134] X. Liu and J. Mellor-Crummey. A Data-centric Profiler for Parallel Programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 28:1–28:12, New York, NY, USA, 2013. ACM.

[135] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overhead. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 183–193, April 2013.

[136] X. Liu and J. Mellor-Crummey. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[137] X. Liu, J. Mellor-Crummey, and M. Fagan. A New Approach for Performance Analysis of OpenMP Programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 69–80, New York, NY, USA, 2013. ACM.

[138] X. Liu, K. Sharma, and J. Mellor-Crummey. ArrayTool: A Lightweight Profiler to Guide Array Regrouping. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 405–416, New York, NY, USA, 2014. ACM.

[139] LLVM Compiler. `http://www.llvm.org`, 2013.

[140] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: Migrating Critical-section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[141] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[142] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Delegated Isolation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 885–902, New York, NY, USA, 2011. ACM.

[143] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 801–810, Berlin, Heidelberg, 2006. Springer-Verlag.

[144] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, June 2005. ACM.

[145] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42nd Annual IEEE/ACM*

International Symposium on Microarchitecture, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[146] P. S. Magnusson, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Washington, DC, USA, 1994. IEEE Computer Society.

[147] A. Malony, S. Shende, W. Spear, C. Lee, and S. Biersdorff. Advances in the TAU Performance System. In *Tools for High Performance Computing 2011*, pages 119–130. Springer Berlin Heidelberg, 2012.

[148] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[149] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 18–29, New York, NY, USA, 2002. ACM.

[150] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '92/PERFORMANCE '92, pages 1–12, New York, NY, USA, 1992. ACM.

[151] C. McCurdy and J. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proceedings of the Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96, March 2010.

[152] S. McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[153] S. A. McKee. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, pages 162–167, New York, NY, USA, 2004. Computer Systems Laboratory, Cornell University.

[154] J. Mellor-Crummey. *Final Report: Correctness Tools for Petascale Computing.* October 2014.

[155] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. Technical Report UR CSD / TR342, University of Rochester. Computer Science Department, 1990.

[156] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *International Journal of Parallel Programming*, 29(3):217–247, June 2001.

[157] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[158] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[159] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, Sept 2008.

[160] G. Moore. Progress In Digital Integrated Electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.

[161] D. Mosberger et al. libunwind Project. `http://www.nongnu.org/libunwind`, 2013.

[162] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, 2007.

[163] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.

[164] G. Nakhimovsky. Debugging and Performance Tuning with Library Interposers. `http://dsc.sun.com/solaris/articles/lib_interposers.html`, July 2001.

[165] R. Narayanan, B. Azisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. MineBench: A Benchmark Suite for Data Mining Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 182–188, 2006.

[166] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 284–295, Washington, DC, USA, 2005. IEEE Computer Society.

[167] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[168] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, June 2007.

[169] C. G. Nevill-Manning and I. H. Witten. Linear-Time, Incremental Hierarchy Inference for Compression. In *Proceedings of the Conference on Data Compression*, DCC '97, pages 3–, Washington, DC, USA, 1997. IEEE Computer Society.

[170] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2005. The Internet Society.

[171] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer Berlin Heidelberg, 1999.

[172] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, May 2006.

[173] Nvidia Corp. CUDA Tools SDK CUPTI User's Guide DA-05679-001_v01, October 2011.

[174] Nvidia Corp. NVIDIA 2011 Global Citizenship Report. `http://www.nvidia.com/object/gcr-energy-efficiency.html`, 2011.

[175] Nvidia Corp. NVIDIA CUDA C Programming Guide Version 4.1, October 2011.

[176] Nvidia Corp. Introduction to GPU Computing. https://www.olcf.ornl.gov/wp-content/uploads/2013/02/Intro_to_GPU_Comp-JL.pdf, 2013.

[177] Nvidia Corp. Nvidia Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler, January 2013.

[178] R. Odaira, J. G. Castanos, and H. Tomari. Eliminating Global Interpreter Locks in Ruby Through Hardware Transactional Memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 131–142, New York, NY, USA, 2014. ACM.

[179] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific Application Performance on Leading Scalar and Vector Supercomputing Platforms. *International Journal of High Performance Computing Applications*, 2006.

[180] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 2–11, New York, NY, USA, 1996. ACM.

[181] Open Solaris Forum. Man solaris - getitimer (2). http://www.opensolarisforum.org/man/man2/getitimer.html, June 2001.

[182] Y. Oyama, K. Taura, and A. Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99)*. Word Scientific, 1999.

[183] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 76–84, New York, NY, USA, 1992. ACM.

[184] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010.

[185] C. S. Park, K. Sen, and C. Iancu. Scaling Data Race Detection for Partitioned Global Address Space Programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 47–58, New York, NY, USA, 2013. ACM.

[186] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[187] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark. *SIGMETRICS Performance Evaluation Rev.*, 38(4):23–29, March 2011.

[188] F. Petrini, D. J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 55–, New York, NY, USA, 2003. ACM.

[189] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995.

[190] D. J. Quinlan. ROSE: compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[191] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 241–, Washington, DC, USA, 2003. IEEE Computer Society.

[192] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[193] A. Rane and J. Browne. Enhancing Performance Optimization of Multicore Chips and Multichip Nodes with Data Structure Metrics. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 147–156, 2012.

[194] J. Reinders. Intel® Architecture Instruction Set Extensions Programming Reference. `https://software.intel.com/sites/default/files/m/9/2/3/41604`, 2012.

[195] J. Reinders. Transactional Synchronization in Haswell. `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012.

[196] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, pages 65–79, 2010.

[197] H. Richard. SQLite. `http://www.sqlite.org/`, 2015.

[198] Rogue Wave Software. ThreadSpotter, Manual, Version 2012.1. `http://www.roguewave.com/documents.aspx?Command=Core_Download&EntryId=1492`, August 2012.

[199] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. ACM.

[200] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15:391–411, November 1997.

[201] W. N. Scherer, III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[202] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl. What Scientific Applications Can Benefit from Hardware Transactional Memory? In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 90:1–90:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[203] F. Schmitt, J. Stolle, and R. Dietrich. CASITA: A Tool for Identifying Critical Optimization Targets in Distributed Heterogeneous Applications. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 186–195, Sept 2014.

[204] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 45–, Washington, DC, USA, 2002. IEEE Computer Society.

[205] M. L. Scott and W. N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 44–52, New York, NY, USA, 2001. ACM.

[206] J. Seward. bzip2. `http://www.bzip.org`.

[207] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[208] SGI. SGI Altix UV 1000 System User's Guide. `http://techpubs.sgi.com/library/manuals/5000/007-5663-003/pdf/007-5663-003.pdf`.

[209] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. Kirby, R. Thakur, and W. Gropp. A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 265–273. Springer Berlin Heidelberg, 2008.

[210] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, Apr. 1988.

[211] J. G. Siek et al. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[212] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 7th edition, 2005.

[213] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[214] M. Sjalander, M. Martonosi, and S. Kaxiras. *Power-Efficient Computer Architectures: Recent Advances*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2014.

[215] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[216] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 813–824, May 2013.

[217] F. Song, S. Tomov, and J. Dongarra. Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and multi-GPU Systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

[218] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 141–150, New York, NY, USA, 2009. ACM.

[219] H. Sun, W.-J. Hsu, and Y. Cao. Competitive online adaptive scheduling for sets of parallel jobs with fairness and efficiency. *Journal of Parallel and Distributed Computing*, 74(3):2180 – 2192, 2014.

[220] Z. Szebenyi, T. Gamblin, M. Schulz, B. De Supinski, F. Wolf, and B. Wylie. Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 640–651, May 2011.

[221] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable Fine-grained Call Path Tracing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[222] N. R. Tallent and J. M. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 229–240, New York, NY, USA, 2009. ACM.

[223] N. R. Tallent and J. M. Mellor-Crummey. Identifying Performance Bottlenecks in Work-Stealing Computations. *Computer*, 42(12):44–50, Dec. 2009.

[224] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary Analysis for Measurement and Attribution of Program Performance. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 441–452, New York, NY, USA, 2009. ACM.

[225] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[226] The Portland Group. PGI Optimizing Fortran, C and C++ Compilers and Tools. `http://www.pgroup.com/`.

[227] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-chip Parallelism. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM.

[228] Unknown. IBM POWER7. `http://www.7-cpu.com/cpu/Power7.html`, 2014.

[229] Valgrind Developers. Valgrind Mailing List. `http://sourceforge.net/p/valgrind/mailman/valgrind-developers/thread/1376648907.3420.16.camel@soleil/`, 2013.

[230] M. Valiev et al. NWChem: A Comprehensive and Scalable Open-source Solution for Large-scale Molecular Simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[231] S. P. Vanderwiel and D. J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, June 2000.

[232] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili. Keeneland: Bringing Heterogeneous GPU Computing to the Computational Science Community. *Computing in Science Engineering*, 13(5):90–95, Sept 2011.

[233] J. Vetter and A. Yoo. An Empirical Performance Evaluation of Scalable Scientific Applications. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 16–16, Nov 2002.

[234] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[235] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu. DrDebug: Deterministic Replay Based Cyclic Debugging with Dynamic Slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 98:98–98:108, New York, NY, USA, 2014. ACM.

[236] S. Wen, X. Liu, and M. Chabbi. Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques, 2015 (to appear)*, PACT '15, 2015.

[237] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[238] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

[239] C.-Q. Yang and B. Miller. Critical path analysis for the execution of parallel and distributed programs. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 366–373, Jun 1988.

[240] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

[241] T.-Y. Yeh and Y. N. Patt. Alternative Implementations of Two-level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 124–134, New York, NY, USA, 1992. ACM.

[242] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

[243] X. Yu, Z. He, and B. Hong. On adaptive contention management strategies for software transactional memory. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 24–31, July 2012.

[244] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity While Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 336–345, Washington, DC, USA, 2005. IEEE Computer Society.

[245] Y. Zhang and E. Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 194–204, New York, NY, USA, 2007. ACM.

[246] Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In *Languages and Compilers for Parallel Computing*, pages 95–109. Springer-Verlag, Berlin, Heidelberg, 2008.

[247] Y. Zhang and J. D. Owens. A Quantitative Performance Analysis Model for GPU Architectures. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 382–393, Washington, DC, USA, 2011. IEEE Computer Society.