

RICE UNIVERSITY

Binary Analysis for Attribution and Interpretation of Performance
Measurements on Fully-Optimized Code

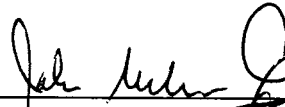
by

Nathan Russell Tallent

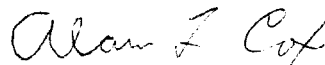
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

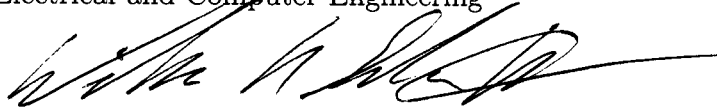
APPROVED, THESIS COMMITTEE:



John Mellor-Crummey, Chair
Associate Professor of Computer Science and
Electrical and Computer Engineering



Alan Cox
Associate Professor of Computer Science and
Electrical and Computer Engineering



William N. Scherer III
Faculty Fellow, Computer Science

HOUSTON, TEXAS

MAY 2007

UMI Number: 1441861

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1441861

Copyright 2007 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Abstract

Binary Analysis for Attribution and Interpretation of Performance Measurements
on Fully-Optimized Code

by

Nathan Russell Tallent

Modern scientific codes frequently employ sophisticated object-oriented design. In these codes, deep loop nests are often spread across multiple routines. To achieve high performance, such codes rely on compilers to inline routines and optimize loops. Consequently, to effectively interpret performance, transformed loops must be understood in the calling context of transformed routines.

To understand the performance of optimized object-oriented code, we describe how to analyze optimized object code and its debugging sections to recover its program structure and reconstruct a mapping back to its source code. Using this mapping, we combine the recovered static program structure with dynamic call path profiles to expose inlined frames and loop nests. Experiments show that performance visualizations based on this information provide unique insight into the performance of complex object-oriented codes written in C++. This work is implemented in HPC-TOOLKIT, a performance analysis toolkit.

Acknowledgments

It has been said that “using the idea of one is plagiarism but using the ideas of many is research.” In this spirit, I would like to thank John Mellor-Crummey and Rob Fowler, both who have been involved in this work in many ways. Particular thanks go to John for committing to see this work to completion during a trying semester and to Rob for bringing Chroma to my attention — the first time to my distress!

If a dedication is needed, it should be to my wife and son who do not take this ‘boring stuff’ too seriously.

Finally, this work has been supported by the Department of Energy Office of Science, Cooperative Agreement No. DE-FC02-06ER25762; and the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory. Experiments were performed on equipment purchased with support from Intel and the National Science Foundation under Grant No. EIA-0216467.

Contents

1	Introduction	1
2	Three Primers	6
2.1	A Profiling Primer's Primer	6
2.2	An HPCTOOLKIT Primer	8
2.3	Related Work	10
3	Recovering Source Code Structure Using Binary Analysis	13
3.1	The Basic Problem and Strategy	16
3.2	Recovering the Procedure Hierarchy	20
3.2.1	Procedure Nesting	20
3.2.2	Procedure Source File and Line Bounds	21
3.2.3	Indirect Procedure Nesting	29
3.2.4	In the Absence of DWARF	32
3.2.5	Macros and Generated Source Code	33
3.3	Recovering Alien Contexts	34
3.4	Recovering Loop Nests	37
3.5	Recovering Groups of Procedures	47
3.6	Normalization	49
3.6.1	Procedures and Loops	49
3.6.2	Alien Contexts	50
3.6.3	Ordering	50
4	From Bloopers to bloop: Implementation	51
4.1	Design and Implementation	51
4.1.1	Processing Instructions	53
4.1.2	Determining Contexts	54
4.2	GNU Binutils and Binary Analysis	58
4.3	Lying Liars	59
5	Performance Visualization Using Dynamic And Static Structure	63
5.1	Deficiencies of Call Path Visualizations	63
5.2	Combining Dynamic Call Paths with Static Structure	65
5.3	Case Studies	66
5.3.1	C++'s STL <code>map</code>	67
5.3.2	Chroma's <code>hmc</code>	75
5.3.3	NAS Parallel Benchmark's CG (UPC Version)	82
6	Conclusions	87

List of Figures

3.1	An object to source code structure mapping.	14
3.2	Typical line map information.	17
3.3	Nested subroutines in Fortran.	23
3.4	Detecting alien code with incomplete DWARF.	25
3.5	Contracting procedures' end lines by instantiating alien contexts. . .	26
3.6	Detecting incorrect class nesting through backward references. . . .	31
3.7	Alien context ambiguity.	34
3.8	Recovering alien contexts within a loop body.	41
3.9	Detecting incorrect loop placement through nesting cycles.	43
3.10	Correcting nesting cycles.	44
4.1	Maximum procedure context nesting.	55
4.2	Erroneous DWARF line maps.	61
5.1	Source code for testing C++'s STL map.	67
5.2	Tau's visualization of STL map example (Figure 5.1).	70
5.3	HPCTOOLKIT's visualization of STL map example (Figure 5.1). . . .	71
5.4	Apple's Shark's visualization of STL map example (Figure 5.1). . . .	73
5.5	Tau's visualization of Chroma.	77
5.6	HPCTOOLKIT's visualization of Chroma (calling context tree). . . .	78
5.7	HPCTOOLKIT's visualization of Chroma (flat).	81
5.8	HPCTOOLKIT's visualization of NPB 2.3 CG.	84

List of Algorithms

4.1	bloop's driver.	52
4.2	determine-context	56
4.3	merge-broken-contexts	57
4.4	matches	58

Chapter 1

Introduction

Performance problems are not created equal: distinguishing between algorithmic ailments, data structure deficiencies, memory hierarchy maladies and instruction stream inefficiencies requires a level of understanding that few developers have. Just as debuggers help navigate crooked code paths, performance tools help reveal performance bugs. To aid the performance analyst, a tool must be able to accurately *characterize* and effectively *summarize* the behavior of a given code. Accurate *characterization* means that a tool must be able to *faithfully* represent, with minimal distortion, the run time behavior of an application. Effective *summarization* means that the tool must accurately summarize large amounts of data in a way that enables the analyst to quickly identify and understand performance problems.

Potent characterization and summarization strategies must consider their target domain. For procedural-style (*e.g.*, FORTRAN 77) scientific codes typified by a few large routines and deep explicit loop nests, loops contain most of the computation and therefore understanding loop performance is a key part of improving application performance. For such programs, Mellor-Crummey, Fowler, *et al.* [36] found that annotating loops with metrics such as CPI (cycles per instruction), memory hierarchy bandwidth and wasted FLOPS and then comparing all the loops in the program as peers was a very effective method for quickly directing the analyst's attention to under-performing regions.

The widespread adoption of modular and object-oriented programming styles has changed the way scientific applications are constructed, but it has not changed the fact that loops are still crucial for understanding performance. For example, Chroma [31], Trilinos [28] and Chombo [32] are all C++-based codes in which key solvers have been abstracted into modular libraries. These codes still have deep loop nests, but the nests are often spread across multiple routines: they may be ‘assembled’ at compile time by inlining object methods, or they may be ‘created’ at run time through dynamically-resolved virtual function calls. However, the composition of many small routines, some of which are dispatched dynamically or instantiated at compile time, creates a very complex dynamic call graph. The effect of modular and object-oriented programming styles is that to understand performance, loops must be understood *in their calling context*. Even so, few existing performance tools provide full calling context information; none that do combine it with loop nests.

Call path profilers assign costs based on calling context. One method for collecting call path profiles is to add source-level instrumentation. Tau [39], a set of widely installed performance tools designed for scientific applications, takes this approach. Tau’s procedure instrumentation increments a count of calls to that procedure, measures the procedure’s inclusive cost — *i.e.* the cost of the procedure including all its callees — and records arcs representing the calling context. While Tau does not typically instrument loops, it might seem that an easy way to merge source code loop information with call path data would be to automatically add loop instrumentation.

However, using static source code instrumentation as a measurement technique typically introduces distortion in two areas. The most commonly observed distortion is that instrumenting small routines and loops results in large levels of execution overhead. Even a small amount of instrumentation added to a short routine can

double its cost (100% overhead); instrumenting loops can increase overhead more dramatically.

The less commonly noted distortion is that source code instrumentation can *change* the measured application by preventing important compiler optimizations. By introducing foreign, side-effect inducing code, instrumentation precludes inlining and interferes with loop transformations such as software pipelining. The net effect of instrumentation can be severe enough that the instrumented code may manifest performance problems that do not actually exist in the non-instrumented optimized code. While it is possible to selectively instrument routines to reduce overheads and allow for inlining, doing so affects the accuracy of call path profiles. In summary, because instrumentation affects not only the speed of the generated code but non-trivially affects the actual object code that is executed, it introduces distortion that jeopardizes the accuracy of both characterization and summarization.¹

In contrast to instrumentation, sampling-based call path profiling can achieve high levels of accuracy while maintaining low overhead [6, 21]. On operating systems that support dynamic linking and library preloading, a profiler can be launched on *unmodified* executables, a significant practical benefit when compiling times for complex applications can be hours. This solves the accurate characterization portion of the problem.

The remaining task is to effectively summarize and present profile data. Most importantly, the data should be correlated to important source code constructs such as loops. However, call path profiles of optimized code ‘miss’ frames where routines have been inlined and they have no knowledge of loops. Moreover, both procedures

¹Binary instrumentation does not prohibit compiler optimizations; *e.g.*, both DynInst [10] and PIN [35] insert instrumentation dynamically while ATOM [41] statically rewrites binaries. However, this method can incur high overheads.

and loops can be transformed by compiler optimizations and consequently not directly correspond to source code.

The challenge, therefore, is to compute a mapping between the optimized object code and the source code structure. Ideally, this information would be recorded by the compiler. However, since compilers do not typically record a summary of their transformations, we propose to infer this mapping using binary analysis. An obvious place to start is the binary’s line map, which maps an object address to its corresponding source file, procedure name and line number. Unfortunately, the line map is insufficient to recover the proposed mapping because in the presence of inlining it is impossible to distinguish between inlined and non-inlined code. This implies that to compute the object to source code map, we must obtain additional information through binary analysis. With this mapping in hand, we can then effectively *merge dynamic calling context information with the static structure of the optimized source code*, making both the dynamic and static scopes ‘first-class’ entities for the purpose of assigning cost metrics.

We therefore intend to defend this thesis: *Combining dynamic calling context information with the static structure recovered from optimized application binaries provides unique insight into the performance of modular codes that have been subjected to complex compiler transformations such as inlining and C++ template instantiation. Moreover, binary analysis of the optimized object code and its debugging sections is sufficient to accurately compute the mapping between the object code and the source code structure that is needed to combine the dynamic and static information.* In support of our thesis, we extend several tools, collectively named HPC-TOOLKIT: 1) We rewrite HPC-TOOLKIT’s binary analysis tool *bloop* to compute an object to source code structure mapping for fully optimized binaries; 2) We extend

HPCTOOLKIT’s correlation tool to combine call path profiles with source code structure by exposing inlined frames and loop nests as cost-inducing entities; and 3) We present visualizations of the correlated data and use them to analyze complex codes.

The rest of this document is organized as follows. Chapter 2 provides further background and justification of our approach. Chapter 3 details our methods for binary analysis, while Chapter 4 discusses important implementation considerations for `bloop`. Chapter 5 describes how call path profiles are correlated with `bloop`’s program structure information. Three case studies argue that the resulting call path visualizations enable the analyst to quickly understand intricate details that would be otherwise obscured or unavailable. Finally, Chapter 6 presents our conclusions.

Chapter 2

Three Primers

2.1 A Profiling Primer's Primer

A serial program can be viewed as series of instructions, only one of which executes at a given time.¹ The current instruction — which can be viewed as a particular location in the program — is often identified by the instruction pointer or IP. At a given time and location, the instruction executes in a certain context. In the most general sense, context is the program's state and includes the full memory configuration. However, in practice we usually restrict our interest to the execution stack; and since each activation record on the stack corresponds to a function call, we often use 'context' to mean calling context, *i.e.*, the chain of function calls from the program's entry point to the current instruction. Therefore, serial programs operate in an environment that can be described by the following three dimensions:

- Time²
- Location (*IP*)
- Context

¹Technically, this is not true on superscalar and out-of-order processors, where multiple instructions may be executed at the same time and out of their original order. However, compilers and processors provide guarantees that this low-level parallelism maintains observational equivalence.

²This dimension may be thought of more generally as any monotonically increasing quantity such as retired instructions or L1 cache misses [26].

Parallel programs contain multiple serial threads of execution and therefore require an additional dimension:

- Thread

Profiling tools must be able to accurately *characterize* and effectively *summarize* the behavior of a given program. Accurate *characterization* means that a tool must be able to *faithfully* represent, with minimal distortion, the run time behavior of the program. The most important components of characterization are the method of measurement and the dimensions over which it is performed. Typical measurement methods are instrumentation and statistical sampling. Commonly measured dimensions give rise to typical profile classes:

- Flat profiles collapse time and context to generate an IP histogram.
- Call path profiles collapse time but expose calling context to create an IP-calling-context ‘histogram.’
- Many parallel tools collapse location and context to create process-time visualizations of inter-process communication.

Effective *summarization* means that a performance tool must accurately summarize large amounts of data in a way that enables the analyst to quickly detect and understand performance problems. One key aspect of this is correlating the summarization with important source code constructs.

As a matter of practice, it is common to collapse the time dimension to scale well on long runs. A typical scalable approach to parallel performance analysis initiates per-process monitoring during program execution and then applies a post-mortem analysis phase to the resulting data set before presenting it to the analyst.

2.2 An HPCToolkit Primer

This work has been done in the context of the HPCTOOLKIT performance tools and our binary analysis methods, in particular, have been implemented in one of its tools known as `bloop` [36]. The following provides a brief background of HPCTOOLKIT.

`bloop` was motivated by the observation that since loops embody most of the computation within scientific programs, understanding loop nests is often critical to improving performance. In particular, treating loops as ‘first class’ entities enables the analyst to quickly compare performance data for all loops in a program as peers, without regard for what file, procedure, or load module they are in. HPCTOOLKIT’s original use of `bloop` was to enrich flat IP-histogram profiles by merging them with the program’s static loop structure information.

`bloop` uses binary analysis to correlate performance data with loops. One motivation for this approach was that multi-lingual applications required maintaining multiple production-level front-ends, something that is difficult and costly. An easy objection is that binary analysis trades one language problem for another: if the source code analyzer must be able to analyze multiple languages, the binary analyzer, if it is to be multi-platform, must support multiple ABIs. The response to this is that ultimately, source code analysis is not enough. For example, scientific codes often link to proprietary math or communication libraries for which source code is unavailable, but which can be very important for understanding an application’s performance. Also, because optimizing compilers typically transform loops to improve performance, the loops that are actually executed may not strictly correspond to the source code.

The original `bloop` recovered loop nests in a procedure by first decoding the procedure’s instructions and creating a control flow graph (CFG). From the CFG, it constructed a loop nesting tree using Havlak’s loop nest recovery algorithm [27]. `bloop` then examined each loop in the tree and used the binary’s line map to recover source line information for each instruction within the loop. To derive loop bounds, `bloop` computed the minimum and maximum source line number within each loop. Similarly, to compute the procedure’s boundaries, it used the minimum and maximum source line over all instructions. To ‘undo’ the effect of loop transformations such as software pipelining and loop-invariant code motion, it used a normalization pass that applied the following two rules until it reached a fixed point:

- whenever a statement instance (line) appears in two or more disjoint loop nests, fuse the nests; and
- whenever a statement instance (line) appears at multiple distinct levels of the same loop nest, elide all instances other than the most deeply nested one.

This strategy was very effective on programs for which the primary target of optimization was loops. However, when we applied `bloop` to object code generated from heavily layered C++ applications, we discovered that procedure and loop bounds were highly inaccurate, leading to wrongly attributed costs and useless results. The basic problem was that procedure transformations such as inlining made the line map information ambiguous; and by not understanding this ambiguity, `bloop` made invalid assumptions. Chapters 3 and 4 present our solutions to these problems.

HPCTOOLKIT’s call path profiler, `csprof` [21], was developed because of the need to understand program costs in context. Some promising work was later done on visualizing the resulting call path profiles, but because it was based on a naïve use of the line map, procedure transformations such as inlining made the result confusing

and difficult to understand. Additionally, no loop information was present. Chapter 5 shows how we have combined `bloop`'s static source code structure with dynamic calling context information to obtain unique insight into the performance of modular codes undergoing complex compiler transformations.

2.3 Related Work

The most widely known call graph profiler is `gprof` [24]. It uses instrumentation (inserted by a compiler or binary rewriter) to count call path arcs from call sites to callees and statistical IP (instruction pointer) sampling to estimate the exclusive time of each procedure. To attribute the cost of a call in context, `gprof` makes the assumption that costs are *context-independent* and uniformly distributes a procedure's cost among each of the call sites through which it is invoked; this attribution assumption is frequently inaccurate [45]. Compaq's `hiprof` [29] call path profiler uses instrumentation of procedure calls and returns to measure the inclusive time of a procedure when called from a particular call site, enabling the computation of context-dependent costs. Tau [34, 39] and Intel's VTune [30], both instrumentation-based tools, also apportion costs in a context-dependent fashion. Tau, however, only distinguishes between contexts no greater than a fixed depth. In programs with small procedures and many procedure calls, instrumentation can cause significant execution time dilation [21]. Tau can dynamically compute compensation factors that approximately account for its own overhead.

A second approach to measurement, motivated by reducing overhead, is statistical sampling [4, 5, 7, 48]. HPCTOOLKIT's `csprof` [21] samples the call stack to create a calling context tree. It employs a special trampoline function to memoize invariant

stack prefixes between samples and collect return counts. Overhead is very low (2-5%) even at a sampling rate of approximately 1000 samples per second.

Unfortunately, despite the utility of call path profiles for modular coding styles, summarization and visualization techniques are still unsatisfactory. `gprof`-style textual call-graph reports are common. These reports are a sequence of text blocks, one for each procedure and in descending order according to the procedure's exclusive time, where a given procedure's text block shows both its callers and callees. For large programs with many routines, these reports are intimidating even to those familiar with the code. Another common approach is to present call path data as a graph, with 'hot' paths emphasized and possibly with controls to hide unimportant nodes and edges [23,37]. While for small programs these visualizations are much easier to understand than `gprof`-style reports, large graphs quickly become unmanageable. As of this writing, Hall's work on call path refinements [25] is one of the best ideas for effectively summarizing large call path profiles, though perhaps the most under-used. In essence, Hall proposed a specialized language for writing path filters: a call path refinement defines a filter that when applied to a call path profile, produces the refined graph with appropriately aggregated metrics. Finally, several tools present calling context trees using a graphical interface analogous to a tree-based view of a file system's directories, with nodes sorted according to inclusive and exclusive cost. This method is a simple but effective summarization technique because inclusive sorts enable the analyst to quickly find and expand the 'hot' paths. HPCTOOLKIT and Apple's Shark [3] have adopted this approach. Tau presents data in the `gprof`, call graph, and tree-based formats.

While the importance of correlating profiling information with source code has been widely acknowledged, most of these summarization approaches only trivially correlate dynamic data with source code using the binary's line map; the one exception

is Tau which can correlate data to top-level loops using loop instrumentation. No tool attempts to account for transformed loop nests or procedures (*e.g.*, missing inlined frames in call path data). Some work has been done on correlating dynamic profile information with transformed source code, but it has been based on detailed compiler information. Waddell and Ashley [46] combined static Scheme source structure with dynamic calling information to visualize the performance of Scheme programs. Their system relied on mappings generated by an optimizing Scheme compiler and focused only on visualizing procedures and expressions. Adve, Mellor-Crummey *et al.* [1] developed a performance visualization environment for the dHPF compiler, which used detailed compiler generated mappings to associate performance information with loops and communication-inducing array references.

To recover source code structure, we employ binary analysis techniques. To our knowledge, no other tool aside from the original version of `bloop` [36] uses binary analysis for this purpose. Tools such as ATOM, DynInst, and PIN are all designed for carefully adding instrumentation to a binary. The ATOM framework [41] provides a simple and general API for statically instrumenting fully-linked binaries. To support dynamic linking, Buck and Hollingsworth developed DynInst [10], an API for generating run time instrumentation. PIN [35], another tool for dynamic binary instrumentation, relies on a just-in-time compiler to dynamically generate a new instrumented instruction stream from a native binary.

Chapter 3

Recovering Source Code Structure Using Binary Analysis

To combine dynamic calling context information with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure. Since the most important elements of the source code structure are procedures¹ and loop nests — procedures embody the actual executable code while loops often consume the bulk of the executable time — we focus our efforts on them. An example of what this mapping might look like is shown in Figure 3.1. In this example, the object to source code structure map is represented as a tree of scopes, where a load module (the binary) contains source files; files contain procedures; procedures contain loops; procedures and loops contain statements; and where scopes such as procedures, loops and statements can be annotated with object code address interval sets. However, for the very same reasons that procedures and loops are important to source code structure, they are also the primary targets of compiler transformations — rendering methods solely based on source code analysis useless.

There are two ways in which a performance tool can account for a compiler's source code transformations: it must either have access to a summary of actions

¹Because of our focus on binary analysis, we use 'procedure' as a synonym for routine, subroutine, or function.

<LM n=".../hmc" base_addr="0x4000000000000000">	<i>load module</i>
<F n=".../hmc.cc">	<i>source file</i>
<P n="doHMC" l="257-449" addr="[0x1eac0-0x21720) ">	<i>procedure</i>
...	
<S l="309-309" addr="[0x1f1b6-0x1f1c6) ..."/>	<i>statement</i>
<L l="311-435" addr="[0x1f460-...) ">	<i>loop</i>
...	
<S l="313-313" addr="[0x1f250-0x1f256) ..."/>	
...	
</L>	
...	
</P>	
</F>	
...	
</LM>	

An object to source code structure mapping represented as a static scope tree expressed in XML. Static scopes include a load module (LM), file (F), procedure (P), loop (L) and statement (S). Procedures, loops and statements are annotated with corresponding object address interval sets.

Figure 3.1: An object to source code structure mapping.

performed by the compiler or it must analyze the resulting object code and attempt to reconstruct this information. Since vendor compilers do not provide the former, we pursue the latter approach: some form of binary analysis. However, to achieve our goal of an object to source code structure mapping, we *do* need some basic object to source code mapping information. Such information is ordinarily found within a binary's debugging section; a common representation for it is DWARF [20, 44].²

The fact that debugging information formats such as DWARF provide a rich language for recording a large number of compiler transformations may seem to obviate the need for complex binary analysis. As an example, DWARF provides a mechanism (DW_TAG_inlined_subroutine) that enables a compiler to describe trees of inlining decisions where each 'node' in the tree associates a source code call site with its

²We use 'DWARF' to refer to both DWARF version 2 and version 3, unless otherwise qualified as DWARF2 or DWARF3, respectively. Version 3, finalized in December 2005, significantly extends version 2 (1993), though the latter is still commonly used.

corresponding object code address ranges in the host procedure.³ Such information about inlining would greatly simplify the job of accounting for procedure transformations, and at the very least, approximate the transformation-history information that we claimed was not available.

In practice, however, most compilers — including those from Intel, PathScale and The Portland Group (PGI) — do *not* generate inlining information, although GCC 4.x does.⁴ However, even if all compilers exploited DWARF’s ability to record information about inlining, it would still not be possible to describe loop transformations using DWARF, for there is no mechanism for doing so. This is understandable given that DWARF intends to describe transformations that directly affect a debugger.⁵ Therefore, because inlining information is not typically available from production compilers — even though the potential has existed for several years with DWARF2 — and because loop nest information is crucial to application performance, it is necessary to analyze both the object code and its associated debugging information to recover source code structure.

Henceforward, we focus our discussion on the DWARF debugging information format which is the *de facto* standard in the Linux world. Even so, because our work is

³DWARF2 permits only one contiguous object code address range, rather than a list of disjoint ranges. Cf. §3.3.8 of [20, 44].

⁴Recent Intel compilers have an option (`-inline-debug-info`) that purports to control whether the compiler “preserve[s] the source position of inlined code” or “assign[s] the call-site source position to inlined code.” We have closely examined the debugging information on some test cases and found no difference in how inlined code was handled.

Some versions of GCC 3.3 do generate inlining information, though a bug impeded generation in versions of 3.4.

⁵It is perhaps debatable whether loop transformations are directly relevant to a debugger or not [9]. Certainly, they are of use to other downstream tools. However, at the very least, for common debugging tasks *that do not focus on debugging performance*, describing procedure transformations and data objects is more important: consider a user who wishes to set a break point in an inlined routine.

It is interesting to note that Silicon Graphics developed some DWARF2 extensions [40] for use with their MIPSpro compilers that describe a loop’s begin points, unrolling factors and software pipeline depth.

only thinly tied to DWARF, using no exceptional descriptive information, it is widely applicable to other contexts. Furthermore, while we target our work to the common languages for scientific computing (Fortran, C, and C++), our object code analysis is very general and adaptable to other languages with similar characteristics. However, our methods are less useful on functional coding styles that employ many dynamically created procedures (‘closures’) and that primarily use recursion to implement repeated computations.

3.1 The Basic Problem and Strategy

A binary’s line map — supported by all debugging formats we are aware of — associates an object code address with a source file, procedure name and source file line number. Excerpts of an actual line map for a procedure named `main` are shown in Figure 3.2. The inlining within `main` is quite complex and includes nested template instantiations; in the excerpts, all object code that derives from source code outside of `main` is marked as *alien*. Observe that every address of the object code maps to a procedure name of ‘`main`’ — even though the object code contains fragments from several *different* procedures. In other words, the line map retains the *original* file and line information (from *before* inlining) but assumes the name of the host procedure (*after* inlining).

Since the original `bloop` [36] (*cf.* Section 2.2) assumed that all object code within a procedure mapped to the same source file and procedure — as might seem to be the case from the line map’s procedure names — it expanded source procedure and loop line bounds whenever it observed a new source line outside the currently accepted bounds. Thus in the example of Figure 3.2, where `main` actually ranges from lines 486-669, the original `bloop` would see a minimum line of 14 (from an

Line Map				Comments	
Address	File ^a	Line	Proc. ^b	Alien?	Difficulty
0x*15550	.../hmc.cc	499	main		
0x*15560	-	97	-	Y	Two <i>different</i> inlined procedures from same file
0x*15570	-	14	-	Y	
0x*15580	-	506	-		
0x*158e0	-	641	-		Out-of-order line numbers
0x*17020	-	527	-		
0x*17030	.../qdp_multi.h	35	-	Y	Two <i>different</i> inlined procs. from an <i>external</i> library
0x*17036	.../qdp_multi.h	57	-	Y	
0x*171f0	-	543	-		Two <i>different, nested</i> template instantiations
0x*171f6	.../singleton.h	456	-	Y	
0x*17200	.../singleton.h	433	-	Y	
0x*172c0	.../bits/stl_tree.h	1110	-	Y	Two <i>different, nested</i> STL instantiations
0x*172d0	.../bits/stl_tree.h	587	-	Y	
0x*173c0	.../objfactory.h	175	-	Y	More nested inlining (STL)
0x*173d0	.../bits/stl_tree.h	579	-	Y	Line # within bounds of host

This table contains a line map for procedure `main` from hmc 3.22.3, part of Chroma [31]. The code was compiled with Intel 9.1 (Itanium) and the object code for `main` ranged between addresses 0x4000000000015340–0x4000000000018140.

The left half of the table shows excerpts of the actual line map for `main` while the right half identifies whether the given instruction derives from source code outside of `main`, *i.e.*, is *alien*. Given that `main` was defined in source file ‘hmc.cc’, how could one determine its source code bounds? (The actual bounds are lines 486-669.)

^aThe symbol ‘-’ indicates the expected file, *i.e.*, ‘.../hmc.cc’

^bThe symbol ‘-’ indicates the expected procedure, *i.e.*, `main`

Figure 3.2: Typical line map information.

inlined procedure earlier in the same file) and a maximum line of 1110 (deriving from an inlined STL procedure). It would therefore report the procedure’s bounds to be 14-1110, inappropriately including many unrelated procedure definitions from source file ‘hmc.cc.’ Since inlined code can appear within loops as well, the effect of this invalid assumption was that both procedure and loop boundaries could be erroneously expanded to include completely unrelated source code, rendering both the structure information and attribution useless in the presence of compiler transformations such as inlining. Even if `bloop` had attempted to consider only those line map entries

that mapped to the same source file as `main`, it would have still faced problems. The minimum line in the excerpt is 14, but that line actually derives from the *same* file as `main`. Since routines either before or after `main` may be inlined into `main`, knowing only the file name does not help.

Clearly, we cannot reliably detect alien code without additional information. If we had both source file *and* line bounds information, it would be possible to distinguish between native and alien code in the line map: any address mapping to a different source file, or to the same source file but outside the given bounds, must be alien. One suggestion might be to ‘carefully’ use the line map to make an initial estimate of the line bounds. However, it turns out that this is not easy:

- *What file defines the procedure?* While in many cases the first instruction in a procedure maps to the host procedure’s source file, it sometimes maps to inlined source code.
- *What are the source line boundaries of a procedure?* In general, no particular instruction is guaranteed to map to the procedure’s begin or end line.

Although the line map’s entries are precisely the information needed to make the step feature in a debugger correctly follow the source code (*i.e.*, file and line), it is difficult for a binary analyzer to use this information to unambiguously identify alien regions within a procedure. If an analyzer *did* know the answers to the two questions posed above, it could at least identify all alien code within a procedure, though it would be insufficient to either partition the alien regions into single procedures or to recover the nested inlining tree. Nevertheless, identifying alien code is an important prerequisite for accurately recovering procedures and loop nests.

Another helpful observation is that source code does not ‘overlap.’ For example, two source procedures do not have overlapping source lines unless they are the same

procedure or one is nested inside the other. This observation extends to structured loops as well. More generally, we can view a source file with lines and columns as a one-dimensional line segment by collapsing its columns and where procedures and loops are located at certain intervals in the segment (their line numbers).

Non-Overlapping Principle of Source Code. *Let scopes x_1 and x_2 have source line intervals σ_1 and σ_2 within the same file. Then, either x_1 and x_2 are the same, disjoint or nested, but not overlapping.*⁶

- $(x_1 = x_2) \Leftrightarrow (\sigma_1 = \sigma_2)$
- $(x_1 \neq x_2) \Leftrightarrow ((\sigma_1 \cap \sigma_2 = \emptyset) \vee (\sigma_1 \subset \sigma_2) \vee (\sigma_2 \subset \sigma_1))$

We can also say (where $x_2 \ni x_1$ means x_1 is nested in x_2):

- $(\sigma_1 \cap \sigma_2 = \emptyset) \Leftrightarrow ((x_1 \neq x_2) \wedge \neg(x_1 \ni x_2) \wedge \neg(x_2 \ni x_1))$
- $(\sigma_2 \subset \sigma_1) \Leftrightarrow (x_1 \ni x_2)$

One implication of this principle is that if we can recover nesting information for procedures, then we can infer some information about source line bounds and vice-versa.

We desire, therefore, to exploit additional DWARF information to enable `bloop` to use the line map more effectively. However, since we want `bloop` to be broadly applicable, our goal is to identify a ‘lowest common denominator’ set of DWARF information generated by all vendor compilers to supplement the line map.

⁶Unstructured programming constructs can give rise to situations that seem to violate this principle. For example, one could create an (irreducible) loop with multiple entry points using unstructured control flow that may be thought of as two overlapping loops. However, in this case, even though the control flow ‘overlaps’, the source code still strictly obeys the given constraints. An *actual* exception is creating an ‘unstructured procedure’ with multiple entry points and a shared exit point using FORTRAN 77’s alternate entry statement. This ‘alternate entry’ forms neither a procedure that is strictly nested nor separate from the host procedure. Fortunately, the alternate entry statement is now deprecated, its use is widely discouraged, and we know of no other source code language used within scientific computing that contains a similar language construct.

3.2 Recovering the Procedure Hierarchy

Given a binary, or more generally a load module, we want to recover accurate source procedure bounds for at least all object procedures in the load module's symbol table. This primarily involves identifying inlined code. However, besides inlining, compilers introduce a variety of other complications. They flatten and reorder a procedure forest (trees of procedures) into a procedure list because ABIs only address procedures by one level of indexing. They may insert data between procedures so that what might appear to be the last instruction of the procedure is actually data and thus is not contained within the line map. Procedures might be split or completely elided from the object code. Also, procedures may be 'duplicated' because of template specializations and instantiations.

3.2.1 Procedure Nesting

While neither C nor C++ support nested procedures, Fortran does allow shallow nesting, to a depth of two. Recovering the procedure hierarchy involves re-nesting the flattened and reordered list that appears in the load module. While it also involves recovering procedure-specific information, this is deferred to the next section.

It turns out that the task of recovering the general hierarchy from DWARF information is fairly easy. DWARF allows procedure descriptors to be nested, where each descriptor is a tree, providing a means for preserving the source code nesting information.⁷ Furthermore, compilers typically generate this nesting information. Note that this information is not esoteric: Pascal — a language for which DWARF was designed to support — supports nested procedures that are within the scope of their parent's local data. In other words, knowing a procedure's nesting is just what a debugger could use to correctly print data values from different procedure nests.

⁷ Cf. §3.3 of [20, 44] and in particular, `DW_TAG_subprogram`.

One complication to this solution is that typically not *all* procedure nesting is indicated by DWARF. In particular, C++ allows classes to be declared within the scope of a procedure, thereby allowing class member functions to be indirectly nested within a procedure. Moreover, this nesting is usually not represented in DWARF. We defer discussion of this to Section 3.2.3 and for now assume programs do contain such code.

3.2.2 Procedure Source File and Line Bounds

Recovering a procedure's source file and line bounds is complicated by compiler transformations. Usually, one expects the first instruction of a procedure to be the start of the prologue and therefore to map to the first source line of a procedure. Similarly, it appears reasonable that, in many cases, the last return instruction — or if there is no return instruction (as could be the case with a tail call) the last instruction — maps to the last statement within the procedure. While one could contrive an example with unstructured control flow that maps the last return instruction elsewhere within the procedure, one might hypothesize that it should still map *within* the source procedure, thereby giving the source file and an approximate end line. However, compilers can schedule instructions before the prologue and relocate the epilogue to somewhere in the middle of the procedure body. Moreover, they can create a frame-less procedure by entirely eliminating the need for an activation record [17] — and then inline other frame-less procedures at the begin and end. Finally, data between functions may make it difficult, without a complete disassembly, to even determine where the last instruction in a procedure is located. Transformations such as these occur frequently enough to render most analyses of optimized modular code useless if they are ignored.

Fortunately, we can again exploit the procedure descriptors within DWARF to obtain *most* of the information necessary for identifying alien code. We have already observed that while DWARF provides a way to record inlining decisions,⁸ the inclusion of this information is the exception rather than the norm. More felicitously, all compilers we are familiar with generate a basic DWARF procedure descriptor that includes the procedure’s name, defining source file and begin line, and low and high addresses.⁹ The first three pieces of data are obviously welcome. Address ranges are important because they enable the analyzer to ignore sections of data in the middle of or at the end of procedures — addresses for which a line map query can return misleading information. Conspicuously absent is a way to describe the source procedure’s end line. It is worth noting again, that this information is tailored for a debugger: while inlining information could enhance usability (*e.g.*, breakpoints in inlined routines), it is not necessary for basic debugger functionality; and end line information is superfluous.

While DWARF provides accurate file names, begin lines and procedure nesting information, this is not yet sufficient to accurately detect alien code. Because a procedure may contain inlined code deriving from before or after it within the same source file, it is essential to have accurate *end* lines. If we combine the nesting, file, and begin line information with the Non-Overlapping Principle, we obtain the following invariants:

Procedure Invariant 1. *A procedure’s bounds are constrained by any parent procedures that contain it.*

⁸*Cf.* DW_TAG_inlined_subroutine in [20, 44].

⁹*Cf.* DW_AT_decl_file, DW_AT_decl_line, DW_AT_low_pc, DW_AT_high_pc, DW_AT_ranges in [20, 44]. The DWARF2 low and high address constructs assume the procedure is contiguous, while DWARF3 allows a series of ranges to describe several non-contiguous blocks. Ranges are inclusive so that the high address is the first address past the last instruction in the procedure.

```

subroutine Y()
  ...
  call x()
  ...
contains
  subroutine x()
    ...
  end subroutine
end subroutine

subroutine Z()
  ...
end subroutine

```

Figure 3.3: Nested subroutines in Fortran.

Procedure Invariant 2. *Let procedure y have sibling procedures x and z before and after it, respectively. Then, y 's begin line is greater than x 's end line and its end line is less than z 's begin line.*¹⁰

These invariants enable an analyzer to infer an upper bound on all procedure end lines except for the last top-level procedure of a source file, whose upper bound is ∞ . Even though a procedure may have an upper bound of ∞ , this can only occur if it is the last procedure in a file, which means that any code inlined into it must come from before it in the same file or from another file. Therefore, assuming we have DWARF descriptors for every procedure, these bounds enable us to detect all alien code if there is no nesting.

The upper bounds we have established thus far, however, are not strictly enough to identify *all* alien code in the case of procedure nesting, even if we assume that all procedures have DWARF descriptors. To see this, consider the procedure hierarchy in Figure 3.3 where Y and Z are two adjacent procedures at the same nesting level. Using the end lines inferred from Procedure Invariants 1 and 2, an analyzer can

¹⁰If the procedures were on a single line, this inference would be incorrect. Practically, we can ignore this possibility. If compilers routinely generated line and column information, this problem could be eliminated.

easily detect all alien lines in Y deriving from procedures that are siblings of Y or that are nested within one of its siblings. However, because Y 's end line bound is known only to be less than Z 's begin line, if another procedure x is both a child of Y and inlined into Y , the analyzer will not be able to detect alien code from x . To avoid this problem, we observe that since Fortran places strict limits on how nested procedures can be defined, an implication of the Non-Overlapping Principle is that the descendants of a procedure form a partition.

Procedure Invariant 3. *Let procedure Y have nested procedures $x_1 \dots x_n$, in that order. Then Fortran nesting implies that the executable code of Y and $x_1 \dots x_n$ forms $n + 1$ ordered, contiguous source code regions.*

Therefore Y should use x 's begin line information to refine its end line bound, enabling it to detect an inlined instance of x .

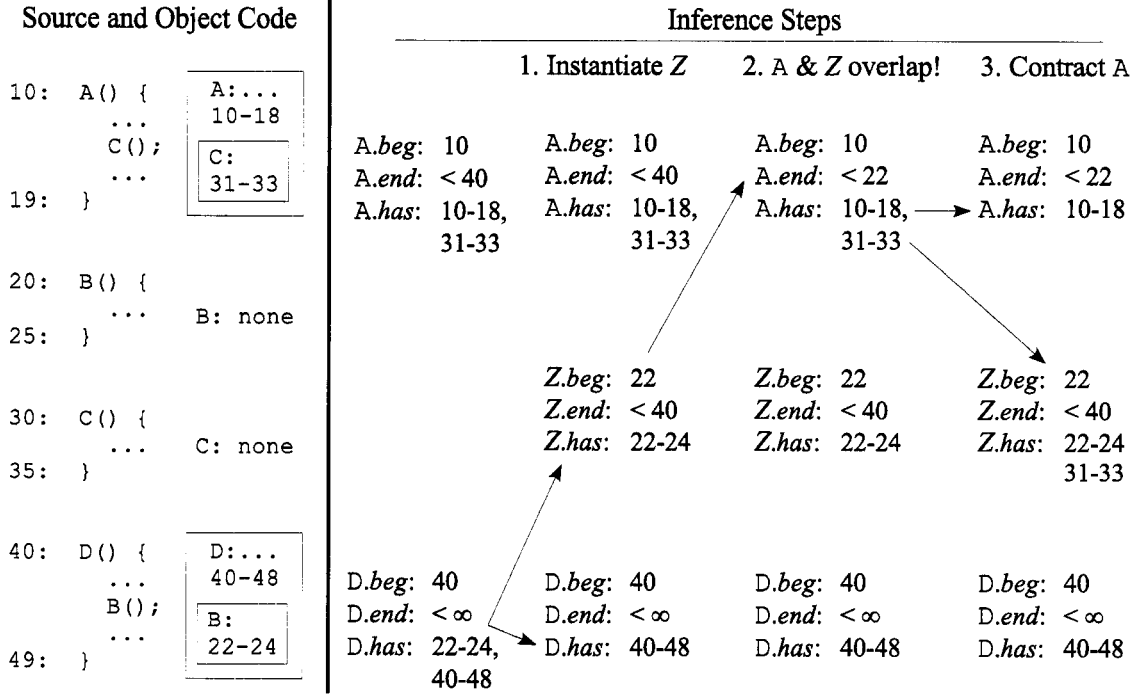
We now have a set of invariants that enable detection of *all* alien code given the presence of basic DWARF information; we refer to the process of bounding procedure end lines as *infer-end-line-bounds*. However, it is often the case that not all source code procedures have associated DWARF descriptors. In particular, if a compiler can inline all direct calls to a procedure and prove that it is never called indirectly — as is often the case with static functions, non-virtual member functions and template instantiations — it need not generate object code or a DWARF descriptor. Consequently, there are often gaps in our knowledge of the source code ‘procedure space’. Consider the example in Figure 3.4. There are four procedures, two that have associated object code (A and D) and two that have neither object code nor DWARF descriptors (B and C). Furthermore, both A and D contain inlined fragments from C and B, respectively. Procedure Invariant 2 provides an end line bound of < 40 on A which is not tight enough to detect the alien code from C. Consequently, it appears that A overlaps both B and C.

Source and Object Code		Inference Steps	
		1. Detect Alien Code	
10: A() {	A:...	A.beg: 10	A.beg: 10
...	10-18	A.end: < 40	A.end: < 40
C();	C:	A.has: 10-18,	A.has: 10-18,
...	31-33	31-33	31-33
19: }			
20: B() {			
...	B: none		
25: }			
30: C() {			
...	C: none		
35: }			
40: D() {	D:...	D.beg: 40	D.beg: 40
...	40-48	D.end: < ∞	D.end: < ∞
B();	B:	D.has: 22-24,	D.has: 40-48
...	22-24	40-48	
49: }		D.alien:	D.alien: 22-24

The left side (the same as in Figure 3.5) shows source code and its corresponding object code; boxes within the object code represent inlining. The right side shows that alien code has been detected in D but not in A.

Figure 3.4: Detecting alien code with incomplete DWARF.

Clearly, it is therefore desirable to contract procedure end lines as much as possible. One way of doing this is by taking advantage of DWARF's C++ class information. DWARF represents a class and its member functions in the same descriptor tree as nested procedures. (Note that a debugger would use this information for printing data or calling class member functions.) Even if every occurrence of a member function has been inlined, that member function may have a 'partial' DWARF descriptor that provides source file and begin line information for its definition, though the descriptor will be incomplete in the sense that it contains no object addresses entries. By 'instantiating' these partial descriptors, source file scopes containing member function definitions will have additional information by which to tighten procedure end line



The left side (the same as in Figure 3.4) shows source code and its corresponding object code; boxes within the object code represent inlining. The right side shows the inference steps for 'instantiating' *Z* and then using its bounds to tighten (actual procedure) *A*'s end line.

Figure 3.5: Contracting procedures' end lines by instantiating alien contexts.

bounds. We name this instantiate-DWARF-descriptors. (Recall that we assume no procedure-scoped classes.)

A second and more involved possibility for contracting procedure end lines is to 'instantiate' known alien regions. Since alien an region ideally corresponds to a distinct procedure we shall also refer to the region as an alien *context* and therefore call the inference process *instantiate-alien-contexts*. As an illustration, consider the situation in Figure 3.5 which has the same source and object code as Figure 3.4. Initially an analyzer must assume that *A*'s end line is less than 40 which would not enable it to detect the alien code from *C*. However, because begin lines are accurate, the analyzer

can infer that alien code exists in D and instantiate ‘procedure’ Z (Inference Step 1), which overlaps with A. What then can it infer from Z ?

Technically, from the analyzer’s perspective, Z could be an instance of A. This would be the case if A’s source bounds actually included lines 10-33, but 1) optimization eliminated lines 22-24; and 2) a version of A was cloned and inlined into D; and then 3) optimization proceeded to eliminate all but lines 22-24.¹¹ However, this would be rather extraordinary and it would be reasonable to assume in most cases that inferred procedures whose begin lines are not ‘close’ represent distinct procedure contexts. Under this assumption, Z would be instantiated and assume a place within the top-level procedure hierarchy (Inference Step 1).¹² This in turn enables the analyzer to remove lines 31-33 from A and combine them with Z (Inference Steps 2 and 3). While lines 31-33 could be split into a new Z' , it is better to simply aggregate regions that the analyzer knows little about as long as they do not overlap known procedures. It should be noted that because alien context region information can change during the analysis of a procedure, ‘instantiation’ of alien contexts should only be done *after* the procedure has been fully analyzed (*cf.* Section 4.1.2).

Because the act of instantiating a procedure introduces new information within a particular file scope, it triggers a propagation pass that checks whether any procedures overlapping the newly instantiated procedure should tighten their bounds — possibly giving rise to more instantiations. Therefore, we would like to establish an upper bound on the number of times that procedure end lines must be updated. Since inferred procedures may only change the end lines of actual (non-inferred) procedures, a propagation pass only needs to consider overlapping with other *actual* procedures

¹¹Recall that often the main purpose of inlining is to expand the known context for analysis, specialization and instruction scheduling.

¹²Technically, Z could be a nested procedure and may not belong at the top level. However, it is still valid to use Z ’s bounds to contract actual procedures’ end lines.

within the same file scope. Moreover, the number of times a procedure can be instantiated is independent of the number of times the procedure is inlined. Therefore, even if a procedure is inlined many times, the first time it is detected and instantiated is the only time it may trigger a propagation pass.¹³ Observe that we can bound the number of procedures that can be inferred for one file by the number of procedures from that file whose DWARF descriptors do not include object code ranges. Let p_i and q_i be the number procedures within file i with and without DWARF descriptors, respectively. Then the number of propagation passes induced by file i can be bounded by $O(q_i)$ where each pass examines at most $O(p_i)$ actual procedures. While the examination of an actual procedure may result in the creation of yet another inferred procedure, the total number of propagation passes for the whole binary is bounded by $Q = \sum_1^n q_i$ (where n is the number of files). If we let p' be the maximum value of p_i over all files, then we can bound the number of propagated changes for the whole load module by $O(p'Q)$.

While we can establish an upper bound on the inference steps required by `instantiate-alien-contexts`, the profitability of using it to refine actual procedures' end lines is suspect. In practice, the most commonly inlined procedures are class functions and template instantiations, both of which are typically defined before being inlined in their ultimate host procedure(s). This means that without `instantiate-alien-contexts`, most alien code can be easily detected by using accurate file and begin lines. Moreover, while `instantiate-alien-contexts` has the potential of improving alien detection, it can also make incorrect inferences (as illustrated in the example where A's bounds could have been 10-33) and hence does not have the desirable property of being 'conservative.' Finally, implementing this process of instantiation and propagation is

¹³Technically, cloning and specialization of procedures might create instances where one version appeared to contain lines 10-12 and another 11-13. As with the validity of instantiating Z , a heuristic would have to determine whether the begin lines were 'close' enough.

non-trivial. Instead of `instantiate-alien-contexts`, we recommend employing `instantiate-DWARF-descriptors`.

Before summarizing our results, we consider two final matters. First, because of compiler-generated template specialization, multiple object code procedures may be generated that map to the same source file and begin line. While we do not regard this as an exception to the Non-Overlapping Principle, it is a complication that must be handled. Second, sometimes procedures are split such that they have multiple symbol table entries and yet appear to have overlapping source code. In such cases, we can fuse the procedures because even though their link names are different, their DWARF names will be identical.

In summary, combining basic DWARF information — procedure nesting, name, source file and begin line — with the Non-Overlapping Principle enables an analyzer to infer highly accurate source code bounds (`infer-end-line-bounds`) as well as detect *all* alien code given DWARF descriptors for every procedure. (Recall that we are assuming the source code contains no procedure-scoped C++ classes.) More practically, when some DWARF descriptors are missing, our methods enable detection of most alien code in typical binaries and do not erroneously classify native code as alien. `instantiate-DWARF-descriptors` is an easy and conservative way to fill in many gaps within the procedure hierarchy. Neither alien names nor nested inlining information can be recovered without additional information.

3.2.3 Indirect Procedure Nesting

C++ permits classes to be declared within the scope of a procedure, thereby allowing class member functions to be transitively nested within a procedure. Recall that to establish procedure nesting, we relied upon the DWARF procedure descriptor tree. Further recall that DWARF represents a class and its member functions in the same

descriptor tree as nested procedures; and that member functions at least have ‘partial’ DWARF descriptors that provide source file and begin line information. While one might expect procedure-scoped classes to be nested within their procedure descriptor, they are usually promoted to the top level of the DWARF tree. Consequently, although nested classes are represented in DWARF as children of their enclosing class, a procedure-scoped outer class is usually not associated with its enclosing procedure.

Assume for the moment that we have DWARF descriptors for every procedure. In this case, re-nesting a procedure-scoped class and its member functions given the procedure hierarchy is trivial since the Non-Overlapping Principle implies that any class member function that has a begin line within a procedure must be nested within that procedure. The challenge, however, is distinguishing class member function code from the enclosing procedure’s code. Suppose a class with one member function is declared within a procedure. This class’s member function is likely to be inlined, causing object code mapping to the member function to be intermingled with code native to the procedure. However, unlike the strict requirements for Fortran nested procedures, the class declaration may appear anywhere in the procedure that a declaration may appear. Since we do not have member function end lines, we need some way of dividing between the source lines for the member function and those for the procedure.

More generally, suppose we have a procedure-scoped class that possibly contains nested class declarations. Assume there are a total of n member functions between the different classes. C++ permits class declarations to appear not only within class scopes, but within class member function scopes, implying that the member functions do not necessarily form n contiguous regions, as was the case with Fortran nesting (Procedure Invariant 3). Since class declarations within class member functions are extremely rare, we shall ignore this problem for the moment and assume that the

	Steps
<pre> <F n="zoo.cpp"> <P n="zoo" l="10-15"> <Alien f="zoo.cpp" ... l="16-50"> ... </Alien> </P> <Class n="..." l="16-25"> ... </Class> </pre>	<p>2. Code use before a declaration</p> <p>1. <i>Instantiate class (incorrectly)</i></p>

Figure 3.6: Detecting incorrect class nesting through backward references.

member functions can be ‘flattened’ into n contiguous regions. This means we can apply Procedure Invariant 2 to obtain end line bounds on the first $n - 1$ member functions. Moreover, these end lines are tight enough to give unambiguous procedure-sized regions.¹⁴ Unfortunately, Procedure Invariant 1 only bounds the n th member function’s end line by the procedure’s end line, implying that there is no clear way to divide the scope of the n th member function and the rest of the enclosing procedure. This problem means that even with DWARF descriptors for every procedure, we cannot necessarily detect *all* alien code within a procedure. However, since procedure-scoped classes are extremely unlikely to contain important code from the perspective of performance, a reasonable approach is to assign code deriving from the ambiguous region after the n th member function to the procedure.

Of course, in the general case, we cannot assume that we have DWARF descriptors for every procedure. Consequently, given a particular class’s DWARF descriptor, we cannot be sure if the class is a procedure-scoped class or a top-level class. Fortunately, it is rare to declare classes within procedures.¹⁵ Nevertheless, we have developed a reasonable strategy (which has not been implemented) that should be able to detect most instances of procedure-scoped classes. We first observe that it is very unlikely for

¹⁴Technically, these regions could overlap data declarations, but this is unimportant.

¹⁵For example, one use for procedure-scoped classes is as a type argument to a template instance that is local to the procedure. However, procedure-scoped classes cannot be used in this way.

a procedure with a procedure-scoped class to be inlined; conversely, it is extremely likely that at least one of the member functions of the procedure-scoped will be inlined. Because of this, a procedure-scoped class that is wrongly instantiated as a top-level class can only result in erroneous alien detection for one procedure, namely its host procedure. Moreover, included with the class information that compilers generate is the begin line of the actual class declaration. Thus, we can employ a *guess and correct* strategy, as depicted in Figure 3.6. First, an analyzer instantiates a given class descriptor with the assumption that it derives from the top-level (Step 1). Then, after processing the procedure (if any) that is immediately before the class *declaration*, the analyzer checks that procedure. If it observes a situation similar to Step 2 where class member functions have been inlined *before* they were declared, it has found an impossibility. An adequate resolution is to nest the class and repeat analysis of the procedure.

3.2.4 In the Absence of DWARF

Sometimes a load module contains procedures without DWARF descriptors. Since this also implies that such a procedure has no entry within the DWARF procedure hierarchy, an analyzer cannot easily use Procedure Invariants 1, 2 and 3 to establish limits on the procedure's bounds.

Assuming that only a small fraction of the object code has no DWARF descriptor, we have had moderate success with simply consulting the begin and end instruction to estimate a procedure's bounds and source file. We have also found that carefully using 'fuzzy' line matching for determining whether a statement is alien frequently improves results. 'Fuzzy' line matching tests for a statement's inclusion within the current procedure's bounds by using an upper and lower bound tolerance factor. If the tolerance factor would include the statement within the bounds, the statement is

deemed to be native and the bounds grow accordingly to include it. Since statements (instructions) are processed incrementally, bounds can also grow incrementally, with different rates, depending on the tolerance factor. See Algorithm 4.4 on page 58 for an example implementation. While this heuristic is not conservative, its results have been quite acceptable.

For load modules without *any* supplemental DWARF information — or its equivalent from another debugging format — prospects for accurately recovering procedure bounds and detecting alien code in highly optimized binaries are bleak. As previously discussed, typical line map information is too irregular to permit simple rules to compute procedure bounds. While it is not likely to be helpful, it is worth noting that *if* accurate procedure bounds are already available, it is easy to recover the procedure hierarchy by using the Non-Overlapping Principle.

3.2.5 Macros and Generated Source Code

Most C and C++ (as well as some some Fortran) codes make heavy use of pre-processing directives and macros. Probably the most commonly used directive is `#include` for including the contents of another file. This directive is typically used to provide procedure and data declarations implemented by other files or libraries. To enable the compiler to relate the preprocessed result with the original source, the compiler's preprocessor emits special `#line` directives that contains a source file and line number. A `#line` directive has the form

```
#line line-number source-file
```

and informs the compiler that the line map information for what follows should be relative to the directive's source file and line number. Besides `#include`, another common directive is `#define`, which is commonly used to define a macro 'function'


```

<F n="main.cpp">
  <P n="zoo" l="10-100">
    A1 <A f="moo.cpp" n="zoo" l="10-13">
      ...
    </A>
  L1 <L l="20-50">
    A2 <A f="moo.cpp" n="zoo" l="10-15">
      ...
    </A>
  ...
</L>
...

```

Figure 3.7: Alien context ambiguity.

such as `MAX(a,b)`. These ‘functions’ are expanded — ‘inlined’ — at compile time. While most macro functions are small expressions, some codes make heavy use of these function macros to implement large bodies of code such as loops. Unfortunately, pre-processors typically do not identify expanded macro functions with `#line` directives. While DWARF does provide a means for describing how macros are expanded, compilers typically do not generate this information. Thus, expanded function macros cannot be detected as alien code.

Similarly, some codes employ their own source-code generators using scripts to generate code that is then fed to a vendor compiler. If a generator fails to include `#line` directives, a compiler cannot correlate the generated code with the source code. Fortunately, there is a natural deterrence against excluding this information because it also means that debuggers cannot follow source code!

3.3 Recovering Alien Contexts

Within a procedure in the object to source code scope tree (*cf.* Figure 3.1), an alien context (or scope) indicates the inclusion of alien code. Figure 3.7 shows an example of two alien contexts, A_1 and A_2 . This information enables one to distinguish

between costs due to native and alien code. At least two additional questions present themselves. First, should we attempt to instantiate alien contexts, thereby filling in ‘missing’ information within the procedure forest? Second, should we attempt to recover alien context nesting?

The first question was discussed in Section 3.2.2 where `instantiate-alien-contexts` was used to improve actual procedure’s end lines; it was answered in the negative. Rather, we argued that it was better to ‘instantiate’ procedures without object code but for which partial DWARF descriptors exist (`instantiate-DWARF-descriptors`).

An alternative to instantiating alien contexts is to partition coarse alien context regions into fine-grained regions, which we call `partition-alien-contexts`. Recall that alien code is detected within a procedure when its source line information maps to a different file or is outside of the procedure’s bounds. Consequently, alien context regions are qualified mainly by source file and may include several procedures implying that it is not possible in general to distinguish between separate instances of inlined procedures. However, observe that because alien contexts are qualified by source file, the Non-Overlapping Principle allows them to be partitioned when it can be shown that they overlap a known procedure boundary from the same file. One simple and conservative way of partitioning an alien context is to use begin line information from all available DWARF descriptors — including any C++ class descriptors — that derive from that context’s file. For example, if n member functions of a class are defined within a single class declaration, then C++ class declaration rules and Procedure Invariant 2 imply that there are at least $n - 1$ unambiguous function-sized regions (*cf.* Section 3.2.3).¹⁶ In general, to implement `partition-alien-contexts`, all the DWARF descriptors for a particular source file can be used to partition alien contexts deriving from that file. Also, since neither the extent nor the final locations of alien

¹⁶It is permissible to ignore any classes that are defined within class member functions.

contexts within a procedure are known until it is fully processed, the best way to partition alien contexts is through a normalization post-pass.

With respect to the second question, we have already noted in Section 3.2.2 that we do not have enough information to recover nested inlining. Recovering nesting actually involves two separate issues: recovering procedure-sized rather than file-sized alien contexts, and then nesting them. As just discussed, file-sized alien contexts can be accurately partitioned, though not necessarily always into procedure-sized chunks. The most problematic issue is nesting.

Because recovering alien context nesting would enable us to accurately fill in sequences of missing frames with call path profiling data, a solution is desirable. An ideal solution is for compilers to generate DWARF inlining information. Another solution is to merge static call graph information with the recovered alien contexts to infer nesting. Such call graph information could be constructed by a source-level tool or a binary analyzer. A binary analyzer would need to analyze each call site in the binary, attempting to disambiguate indirect calls (*e.g.*, those made through registers) using data-flow analysis. It should be noted that there is no correlation between instruction order and nesting.

Without call site information, alien context recovery may encounter ambiguous information that cannot be resolved. The example in Figure 3.7 shows two alien contexts A_1 and A_2 with overlapping bounds, where A_2 is embedded within loop L_1 . If there were no alien contexts, loop normalization (*cf.* Section 3.6.1) would apply a version of the Non-Overlapping Principle, and merge the code outside the loop with the overlapping instances inside it, accounting for loop-invariant code motion. However, without call site information, we cannot distinguish between 1) one distinct call site within the loop, where some of the inlined code was loop invariant; or 2)

two distinct call sites where some of the code from the first call site (A_1) was entirely eliminated.

A number of details with respect to managing alien context detection and recovery are discussed in Section 4.1.2.

3.4 Recovering Loop Nests

Loops form the computational core of most scientific applications. Therefore, given the object code for a certain procedure, we wish to recover loop nests with accurate source line bounds.

This task can be broadly divided into two components: 1) analyzing object code to find loops and 2) inferring a source code representation from them. To find loop nests within the object code, our **bloop** analyzer first decodes the instructions in a procedure. Then, using the OpenAnalysis infrastructure [42], it reconstructs the control flow graph (CFG) and then computes the tree of strongly connected regions using Havlak’s algorithm [27] for recovering loop nests. Within the tree, each node corresponds to a basic block of the CFG and each cyclic reducible (single-entry) region corresponds to a loop where the target node of the backward branch is treated as the loop header. Irreducible (multi-entry) regions are examined for loops and can themselves optionally be treated as loops.

Given this tree of object code loops, **bloop** then combines each loop with source file information to reconstruct a close approximation of the corresponding source code loop nesting tree. This involves identifying alien code (using the techniques of Section 3.2), inferring loop bounds, and reversing compiler transformations such as software pipelining that create multiple object code loops that map to the same source lines. However, in contrast to the problem of recovering the procedure hierarchy, there is

no additional DWARF information available to the analyzer for forming a skeletal solution: DWARF has no constructs for representing loops. Moreover, within fully optimized binaries, the object code is a swirl of instructions, possibly from different contexts, that is the product of loop transformations such as loop-invariant code motion, loop unrolling, loop distribution, loop fusion and software pipelining. Since we have no definite information about where loops begin or end within source code, the only thing we can assume is that accurate procedure bounds enable us to detect alien code. Given this situation, any algorithm must approximate a solution; the challenge, therefore, is to develop a set of heuristics that are general enough to apply to all vendor compilers but powerful enough to produce accurate results.

It is not immediately clear where to begin searching for solid ground. One obvious observation is that nested source code loops also obey the Non-Overlapping Principle. Two immediate inferences are that:

Loop Invariant 1. *A loop's bounds are constrained by any outer loops that contain it as well as its containing procedure.*

Loop Invariant 2. *Let loop y have sibling loops x and z before and after it, respectively. Then, y 's begin line is greater than x 's end line and its end line is less than z 's begin line.¹⁷*

These two observations would be powerful tools if we had some prior knowledge of loop bounds. Another limitation is that while they accurately describe source code, they do not elegantly describe the transformed loops that the object code is generated from. For example, when loops are split while partitioning the iteration space, line map information will suggest that the object code contains two sibling

¹⁷If the loops were on a single line, this inference would be incorrect. Practically, we can ignore this possibility. If compilers routinely generated line and column information, this problem could be eliminated.

loops whose bounds *do* overlap. This fact means that care is needed when employing these invariants.

A third observation that is just a reformulation of what loop invariant code motion guarantees *not* to do is that: *A computation at level l will (usually) not be moved into a loop that is at a nesting level deeper than l .* (We adopt the convention that top-level loops have a nesting level of 1 and top-level statements a level of 0.) This observation presents an opportunity for developing a heuristic to actually estimate loop bounds. For example, given accurate procedure bounds, we could scan through all the non-alien statements within a particular loop and compute a minimum and maximum line number, which we call the min-max heuristic.

As one might expect, there are complications, most significantly from ‘forward substitution.’ A Fortran *statement function* — a single statement function definition within a procedure that appears after type declarations but before executable statements — may be used within a loop; these statement ‘functions’ have no corresponding DWARF descriptor. As discussed in Section 3.2.3, C++ permits classes to be defined within procedures; and the best end line bound for the last member function is the procedure’s end line bound. In either case, as the statement function or last member function is inlined, a compiler typically associates the expanded code with the definition rather than the host location. Consequently, such a forward substitution that appears in a loop will *not* appear to be alien to the procedure, even though it *is* alien to the loop. As a result, a forward substituted statement appearing in the first loop of a procedure will cause the begin line of this loop to expand and include all statements prior to the loop. While inordinate expansion of other loops can be prevented using Loop Invariant 2, significantly erroneous bounds for the first loop is problematic enough: we have observed a situation where such distortion caused very inaccurate results for the first loop in an important procedure.

To prevent this problem, we would like some mechanism of estimating the begin line of a loop. When loops are compiled to object code, the loop header test is typically translated into a conditional backward branch that, based on the result of the loop test, returns to the top of the loop or falls through to the next instruction. Consequently, we expect a loop’s backward branch to retain the source line of the loop condition, and therefore the loop header. Thus, it appears reasonable to consult the source line information of this backward branch instruction to approximate the loop begin line. Expectation and reality can often be disjoint, but it turns out that compilers usually construct the line map in this manner (excepting GCC 3.4.x). Since no analogous ‘backward substitution’ problems exist to threaten loop end bounds, we can modify the simple min-max heuristic to form the **bbranch-max** heuristic for computing loop begin and end lines: the loop begin line can be approximated using information from the backward branch; and the best loop end line is the maximum line after all alien and non-loop lines have been removed.

Unfortunately, this modified heuristic is still more fragile than we would like. For one, we would like to handle GCC 3.4.x. A less common issue is that loops written using unstructured control flow often have a loop test and backward branch at the end of the loop — which is not surprising since object code itself uses unstructured control flow constructs. Since it is not possible from the object code to distinguish between structured and unstructured control flow, the **bbranch-max** heuristic will assign a loop begin line larger than it should be. This presents us with a serious practical dilemma because the analyzer will eject lines that actually belong in the loop, having the two-fold effect of distorting metric attribution (*e.g.*, assigning costs to a non-loop statement that should belong to the loop) and annoying the analyst.¹⁸

¹⁸As one might suspect, the relative severity ratings of these effects varies with the analyst and his mood!

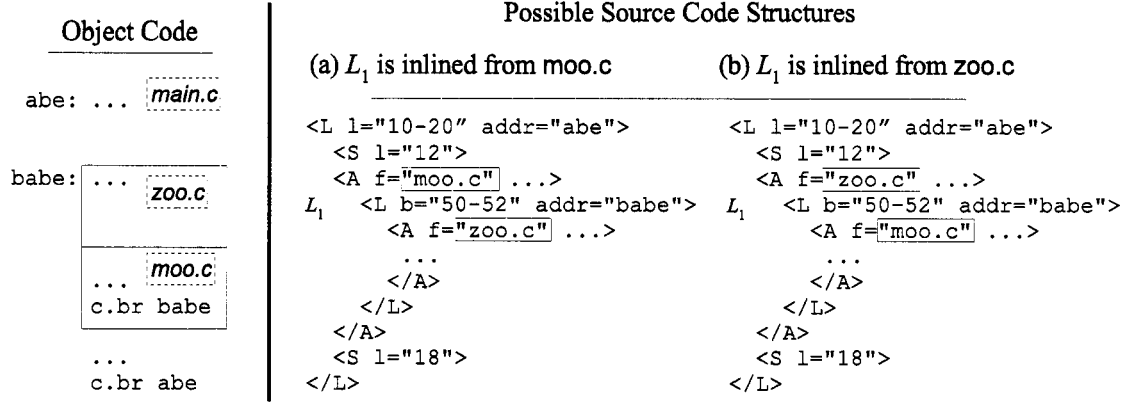


Figure 3.8: Recovering alien contexts within a loop body.

We have therefore experimented with ‘fuzzy’ line matching (introduced for procedures in Section 3.2.4) to form the `bbranch-max-fuzzy` heuristic. For loops, we found that a begin line tolerance of 5 was small enough to usually prevent a loop from growing past its actual begin line, but large enough to reach the begin line. (*Cf.* Algorithm 4.4 on page 58; the end line tolerance is ∞ to assign the maximum line within the loop as the end line.) Of course, it is possible to confound this heuristic with, *e.g.*, Fortran statement functions located within the tolerance factor of the loop. However, because statement functions must appear after type declarations and before executable statements, the Non-Overlapping Principle bounds a loop’s begin line both by the first executable statement and by the procedure’s begin line.

While `bbranch-max-fuzzy` handles situations with simple inlining quite well, a more severe problem remains. The first part of the problem is simply observing that loops themselves may be inlined. Because loops are typically so important, when recovering loop nests within a procedure, it is imperative to ensure that alien or native loops are assigned to the appropriate alien or native context, respectively. Therefore, the goal of loop recovery is not simply one of computing loop bounds where there is

no definite meta-information, but recovering a loop's line bounds *and* its enclosing *procedure context* without meta-information — where the enclosing procedure context refers simply to the loop's enclosing alien or procedure scope within the scope tree.

The difficulty is that **bbranch-max-fuzzy** always estimates a loop's begin line and procedure context from an instruction associated with the loop's *header*, specifically its condition test. In doing so, it implicitly assumes that the procedure context for that instruction is the same context as other instructions from the loop, including the loop *body*. The impact of this assumption is illustrated in Figure 3.8. The object code in the figure contains two loops, one beginning at **abe** and the other at **babe**. Assume that loop **abe** is correctly located and that we are in the process of determining the procedure context of loop **babe** (L_1). Clearly **babe** is an inlined loop — and we can detect this — but it is not clear whether it was inlined from file **moo.c** or **zoo.c**. If, as Case (a) depicts, the former is true, then **bbranch-max-fuzzy** will correctly choose **moo.c** as the alien procedure context because the backward branch is associated with that file. If on the contrary, loop **babe** actually derives from **zoo.c**, then **bbranch-max-fuzzy** will still infer the result in Case (a) when it should have inferred Case (b). For **bbranch-max-fuzzy**, this situation is inherently ambiguous; in general, it may assume both an incorrect loop begin line and an incorrect procedure context. This error will be compounded if the loop contains other loops, ultimately resulting in both inaccurate cost correlations and incoherent visualizations. Moreover, within object-oriented C++, it is *very* common for loop headers to contain inlined condition tests.

The example in Figure 3.8 was contrived to be especially wicked; in fact, it is fundamentally ambiguous. However, the more common problem is how to distinguish between a loop deriving from an alien context (and which itself may have alien loops)

<pre> <F n="main.cpp"> <P n="init" l="145-199"> A₁ S₁ <S l="82-82"/> L₂ <L l="83-83"> S₂ <S l="83-83"/> A₃ S₃ <S l="158-158"/> ... </pre>	<table> <tr> <th data-bbox="1104 182 1185 220">Steps</th><th data-bbox="1185 182 1476 220"></th></tr> <tr> <td data-bbox="1104 252 1347 294">1. <i>Find alien scope</i></td><td data-bbox="1185 252 1476 294"></td></tr> <tr> <td data-bbox="1104 315 1445 357">2. <i>Locate loop (incorrectly)</i></td><td data-bbox="1185 315 1476 357"></td></tr> <tr> <td data-bbox="1104 420 1315 462">3. <i>Self nesting!</i></td><td data-bbox="1185 420 1476 462"></td></tr> </table>	Steps		1. <i>Find alien scope</i>		2. <i>Locate loop (incorrectly)</i>		3. <i>Self nesting!</i>	
Steps									
1. <i>Find alien scope</i>									
2. <i>Locate loop (incorrectly)</i>									
3. <i>Self nesting!</i>									

Figure 3.9: Detecting incorrect loop placement through nesting cycles.

and one that only contains alien contexts within its header or body. Our solution to this problem, is to *guess and correct*, and is related to the design decisions elaborated in Section 4.1. In brief, the analyzer processes instructions within a loop one-by-one; and for each instruction it determines that instruction’s procedure context and its source line location within that context. Figure 3.9 shows a partially reconstructed procedure where alien scope A_1 has been identified (Step 1) by using the source line information for the instruction corresponding to S_1 . When the analyzer processes the loop header (S_2) for L_2 using `bbranch-max-fuzzy` (Step 2), it must determine whether the source line loop should be located in the current procedure context, a prior context (which would imply the current context is alien), or a new alien context. In this case, thanks to S_2 , the analyzer ‘guesses’ that the loop header should be located within the current alien procedure context A_1 . The analyzer next processes S_3 (Step 3), which it determines must be alien to the current procedure context A_1 , resulting in the new alien context A_3 . However, because A_3 ’s bounds are within `init`’s bounds, this implies that `init` is inlined inside of itself, which is nonsense. This shows that the guess at Step 2 was wrong.

This observation, which is another implication of the Non-Overlapping Principle, can be reformulated more generally as follows:

<i>Before</i>	<i>After</i>
<u><F n="main.cpp"></u>	<u><F n="main.cpp"></u>
<P n="init" l="145-199">	<P n="init" l="145-199">
A_1 	
<S l="82-82"/>	<S l="82-82"/>
	
L_1 <L l="83-83">	<L l="158-158">
	
S_2 <S l="83-83"/>	<S l="83-83"/>
	<A>
	
S_3 <S l="158-158"/>	<S l="158-158"/>

To correct the nesting cycle of Figure 3.9, 1) un-nest L_1 ; 2) update its bounds with S_3 ; and 3) replicate A_1 's context for S_2 .

Figure 3.10: Correcting nesting cycles.

Procedure Invariant 4. *Let L be a loop nest rooted in an alien scope C_a . Furthermore, let L have loop levels $1 \dots n$. Now, let s be the deepest occurrence of a statement at level m ($1 \leq m \leq n$) that clearly belongs in a shallower procedure context C' . Since C' is a shallower procedure context, it must be a parent of C_a which implies that C' is nested within itself, which is impossible.*

When an impossibility such as this is found, the analyzer, knowing that L was mislocated, can correct the situation by relocating all levels of L from C_a to within C' . An example for Figure 3.9 is shown in Figure 3.10. In this case, L_1 is un-nested one level, which places it within the correct procedure context and its bounds are updated to include S_3 . S_2 remains nested in L_1 , but A_1 's context must be replicated to correctly represent it. Observe that fuzzy matching can be a useful heuristic for expanding the the begin line of L_1 . Correcting nesting cycles is discussed in more detail in Section 4.1.2 and Algorithm 4.3 on page page 57.

Combining `bbranch-max-fuzzy` with nesting cycle correction enables a binary analyzer to accurately handle most inlining within loop nests. However, there is still another thorny issue with respect to recovering loops in the vicinity of alien contexts.

Consider the case where a loop is located within an alien context. `bbranch-max-fuzzy` relies on both an accurate loop begin line and procedure bounds to detect code that is alien to the procedure *and* the loop (forward substitution). Recall that we have little ability to distinguish between distinct alien procedure contexts deriving from the same source file. This means that while processing a loop located within an alien context, we have little ability to detect alien code unless it derives from a different file. Even if we assume an accurate loop begin line, a routine from the same file but defined afterward could be inlined into the loop, causing the `bbranch-max-fuzzy` heuristic to improperly expand the loop’s end line. While sibling loops will constrain the loop’s bounds (Loop Invariant 2), this does not help the case of a single loop, which is much more likely to be found inlined than several sibling loops. Given our limited information, there is little that can be easily done. We have experimented with context-sensitive fuzzy matching that makes loop bounds matching more conservative when located within an alien scope. The effects can be complex, because a loop may initially appear to be within an alien context (by backward branch information) but later emerge as a native loop. However, in practice, we have found that tightening the loop end line matching tolerance from ∞ to 20 produces good results (*cf.* Algorithm 4.4 on page 58).

Before concluding this section, we consider two special cases. First, compilers can both remove and add loops to object code. One way a compiler removes a loop is by completely unrolling it and replacing it with an equivalent computation. Such a loop might perform a trivial computation or it might be the inner loop of a loop nest. In contrast to loop removal, a compiler may *introduce* a new loop, *e.g.*, to implement Fortran 90 array or array triplet notation. These two transformations mean that in some cases, an analyzer will be unable to recover an actual source loop, while in other cases it will infer a loop where there is none. However, from the perspective of the

analyst, this is actually a very *desirable* property. Identifying (full) loop unrolling and discovering implicit loops with binary analysis highlights its unique ability to reveal what actually is executed. If, *e.g.*, careless use of Fortran’s array notation induces an unexpected loop — such as a copy when passing a slice as an argument — this is important to know about!

The second special case is that procedures are not always contiguous, such as when compilers place data in the middle of routines. While DWARF address ranges may indicate such non-contiguity, because loop recovery includes basic instruction decoding and creation of the CFG, data will be gracefully ignored.

In summary, the methods of this section enable a binary analyzer to identify loop bounds and procedure contexts with a large degree of precision for very complex object code. Implementation of the **bbranch-max-fuzzy** heuristic and nesting cycle correction is very effective and in practice we often are able to precisely identify loop bounds. These methods, however, are incomplete because we have not yet accounted for loop invariant code motion and loop transformations such as software pipelining. Because of this, the same line instance may be found both outside of a loop and within it (*e.g.*, partial invariant code motion) or there may be duplicate nests that appear to be siblings (*e.g.*, iteration space splitting). To account for compiler loop transformations, we use the normalization passes described in Section 3.6.1.

Our methods for loop recovery have resorted to heuristics such as **bbranch-max-fuzzy** for recovering loop nests. In practice these methods recover very accurate loops, though the potential exists for pathological cases. Because of this, it is important to observe that if the procedure context of a loop is determined incorrectly, the effects of this error are limited at most to *one* procedure, namely the host procedure. Moreover, while **bbranch-max-fuzzy** does makes an assumption (line map information for the backward branch) about typical compiler behavior to identify forward substitu-

tion, it is primarily based on the observation that moving statements into loop nests is profitable neither from a machine-independent nor machine-specific perspective. This suggests that the heuristic is unlikely to be invalidated by a future compiler optimization.

A limitation of the methods of this section is that we have inferred loop constructs through careful use of line-based information as opposed to any sort of source loop descriptor. While in many cases this works extremely well, more complex loop transformations may lessen their accuracy. For example, automatically parallelizing Fortran compilers often attempt to interchange loops, meaning that a certain loop will be moved more deeply into a loop nest. If loop begin lines could be recovered with perfect accuracy, then the analyzer could detect that an outer loop was nested within an inner loop. However, because `bbranch-max-fuzzy` cannot assume begin lines are completely accurate, it will use Loop Invariant 1 and expand the bounds of the new outer loop, creating what appears to be (at least) two loops with the *same* begin line.

3.5 Recovering Groups of Procedures

Within object-oriented code, groups of procedures are often important. For example, C++ classes contain member functions that collectively implement all the operations related to a certain abstraction. Application developers often wish to know what the cost of these abstractions are. Placing member functions in a ‘first-class’ group enables a correlation tool to attribute performance data to all the functions collectively as well as individually. In other words, the effect of inlining small and commonly used procedures is to distribute the costs of these abstractions to their caller; ‘instantiating’ the procedures (and maintaining object code links) enables these costs

to be aggregated, thereby providing a way to gauge the effectiveness of a compiler at minimizing the cost of such abstractions.

Recall that DWARF represents a C++ class and its member functions in the same descriptor tree as nested procedures. Therefore, all member functions of a class are naturally grouped as children of the class, making it trivial to create groups of member function names. If all instances of a member function happen to be inlined, it has a ‘partial’ DWARF descriptor which still has file and line information. Moreover, alien code within a procedure is identified by its file and line of origin. An analysis tool can therefore associate DWARF member function descriptors (whether partial or full) with alien contexts.

However, this involves a number of complexities. If a coarse (‘file-sized’) alien context contains inlined member functions from two different classes, then that one context — along with its associated performance metrics — will be associated with both classes! An approximate solution to this problem is to use DWARF class member function descriptors to partition coarse alien context regions into function-sized chunks as discussed for `partition-alien-contexts` in Section 3.3.

Another use for procedure groupings is to expose the costs of the various instantiations of a template. Instantiations of templated functions are treated as procedures from the perspective of DWARF descriptors. Since non-inlined template instantiations have DWARF descriptors that map to the same begin line and directly overlap each other, groups can be inferred for them. In contrast, inlined template instantiations are analogous to (fully) inlined functions and have no DWARF descriptors. It is worth noting that this method only groups *compiler* generated instantiations for one specific template; in particular, it cannot group several partially specialized templates together because they map to different, non-overlapping source lines.

Since templated classes are classes, compilers typically record class member function information for each instantiated class. In this case, the above methods for a non-templated class can be employed. However, care must be taken since the DWARF class descriptors are unrelated to each other. To avoid considering the same template class multiple times, the DWARF class descriptors should be grouped in some manner using the Non-Overlapping Principle (as was done with templated functions) applied to the first class member function.

In summary, we have identified some ways in which functions can be grouped together to represent abstractions such as C++ classes and templates. These groups enable performance metrics that would otherwise be distributed to be aggregated and charged to the abstraction itself. We suspect that for common and regular coding practices a relatively simple application of our class-based heuristics would yield significant insight. We have not implemented them, however, partly because of the difficulties of using GNU Binutils (*cf.* Section 4.2).

3.6 Normalization

3.6.1 Procedures and Loops

As explained in the closing of Section 3.4, it is necessary to ‘undo’ the effects of compiler loop transformations to remove duplicate line instances. To do this we develop normalizations that are additional applications of the Non-Overlapping Principle.

The first and most important loop normalization, which we call *coalesce-duplicate-statements*, is the appropriate modification of the normalization given in Section 2.2:

- whenever a statement instance (line) appears in two or more disjoint loop nests, fuse the nests *but only within the same procedure context*; and

- whenever a statement instance (line) appears at multiple distinct levels of the same loop nest (*i.e.*, *not crossing procedure contexts*), elide all instances other than the most deeply nested one.

The second normalization, **merge-perfectly-nested-loops**, merges perfectly nested loops with *identical* bounds.

coalesce-duplicate-statements should be applied before **merge-perfectly-nested-loops** because it can create perfect loop nests. When merging statements and loops it is important to also merge their associated object address range intervals.

3.6.2 Alien Contexts

As discussed in Section 3.3, **partition-alien-contexts** can partition file-based alien contexts into more fine-grained contexts using begin line information for all DWARF descriptors that derive from that file.

3.6.3 Ordering

See Algorithm 4.1 on page 52 for an example normalization phase. First, loop normalizations should be applied in the order presented. After this, alien contexts (which may contain loops) may be partitioned with **partition-alien-contexts**.

As artifacts of other normalizations, empty scopes may remain within the scope tree. We find it convenient, therefore, to apply a final normalization pass, **remove-empty-scopes**, to remove any empty scope that is associated with an empty object code address range. Note that a procedure or loop may be empty in the sense that it contains no statements that map to known source lines, but has a non-empty address range. Since profiling data could map to one of these ‘empty’ scopes for which at least a small amount of information is known, it is desirable not to remove them.

Chapter 4

From Bloopers to bloop: Implementation

We have implemented the strategies presented in Chapter 3 within HPCTOOLKIT's `bloop` tool. We currently support several Linux ABI's including x86-Linux, x86-64-Linux, Itanium-Linux and MIPS64-Linux. We support debugging information generated by at least GNU GCC (versions 3.x and 4.x), Intel (versions 8.x and 9.x), PathScale (versions 2.x and 3.x), and The Portland Group (PGI) (versions 6.x).

While our prior discussion suggests the outlines of an implementation, there are a number of important design and implementation issues that remain. These are discussed in the following sections.

4.1 Design and Implementation

The basic algorithms for `bloop` and its helper functions are presented in Algorithms 4.1, 4.2, 4.3, and 4.4.

Algorithm 4.1 shows the main driver, which can be divided into three parts. The algorithm takes a load module and returns the object to source code structure map in the form of a scope tree. The first step is to form a skeletal representation of the procedure hierarchy by consulting the DWARF procedure descriptors for the procedures contained in the binary's symbol table. Procedure representations for partial DWARF descriptors may also be instantiated (`instantiate-DWARF-descriptors`). Second, for each object code procedure, the analyzer creates a control flow graph and

Algorithm 4.1: bloop's driver.

Input: A load module lm (with DWARF debugging information)
Result: Σ , lm 's object to source code structure map (*cf.* Figure 3.1)

let $\Omega : recovered-procedure \mapsto object-procedure$ be the object code map
let $\Delta : object-procedure \mapsto DWARF-descriptor$ be the descriptor map
let $\Lambda : address \mapsto \langle file-name, proc-name, line \rangle$ be the line map

// Infer skeletal structure of Σ
instantiate-DWARF-descriptors (using descriptors in Δ without object code)
foreach *object-procedure* p_Ω in Ω *s.t.* $d \leftarrow \Delta(p_\Omega) \neq \text{NIL}$ **do**
 Create source code representation p_Σ for p_Ω within Σ (using d 's info),
 locating p_Σ within appropriate source file and procedure scopes.
end

Estimate representation for every procedure p_Ω in Ω *s.t.* $\Delta(p_\Omega) = \text{NIL}$

// Infer alien code and loop nests for each procedure in Σ
foreach *procedure* p_Σ in Σ *s.t.* $p_\Omega \leftarrow \Omega(p_\Sigma) \neq \text{NIL}$ **do**
 Determine loop nests for p_Ω by computing the strongly connected regions
 tree T induced by the control flow graph CFG of p_Ω
 foreach *basic block* b in T (*preorder traversal*) **do**
 let es_Σ be b 's immediate enclosing scope (loop or procedure)
 if b is a loop header node **then**
 let $\sigma \leftarrow \Lambda(i)$ for backward-branch i
 $es'_\Sigma \leftarrow \text{determine-context}(es_\Sigma, \sigma)$
 Create a source code loop l_Σ located within es'_Σ
 $es_\Sigma \leftarrow l_\Sigma$
 end
 foreach *instruction* i in b **do**
 let $\sigma \leftarrow \Lambda(i)$
 $es'_\Sigma \leftarrow \text{determine-context}(es_\Sigma, \sigma)$
 Create a statement scope s_Σ for σ within es'_Σ
 end
 end
end

// Normalize Σ
foreach *procedure* p in Σ **do**
 coalesce-duplicate-statements within p (*cf.* Section 3.6.1)
 merge-perfectly-nested-loops within p (*cf.* Section 3.6.1)
 partition-alien-contexts within p (*cf.* Section 3.3)
end

remove-empty-scopes within Σ (*cf.* Section 3.6.3)

constructs a tree of strongly connected regions where each node is a basic block. Then, the analyzer makes a preorder traversal (a parent before its children) over each basic block in the tree, creating loops for blocks that represent loop headers and locating each statement within its containing loop (if any) and procedure context. Finally, a series of normalizations are applied to the resulting scope tree.

We now consider further implementation details.

4.1.1 Processing Instructions

Given a basic block, `bloop` considers each instruction in turn, creating and locating a corresponding statement within the appropriate loop and procedure context before considering the next instruction. The disadvantage of this ‘eager’ strategy is that `bloop` may incorrectly locate the statement and discover only later that it must correct this mistake (as was shown in Figure 3.9). Because the object code is often a swirl of instructions from different procedure contexts, it may seem that considering groups of instructions within a loop would provide a better indication as to where the loop should be located. For example, one strategy might be to defer an instruction’s location until a group of instructions can be located at once by placing it in a ‘lazy location buffer.’ This location buffer would wait until it had ‘enough’ information to locate a group of instructions, and then perform a ‘lazy’ location. The problem with this approach is the same as its motivation: instruction sequences can be significantly reordered and it is not clear either how to reasonably decide when a location decision can be made or how to bound the computation performed by the lazy buffer. In fact, it is sometimes not obvious where a loop should reside until inner loops have been processed; sometimes it is fundamentally ambiguous (*cf.* Figure 3.8). For example, in the presence of heavy inlining, a loop may appear to be located within one of several alien contexts until a statement that clearly belongs to the host procedure is found in

a deeper loop nest. The implication is that at the very least, inter-basic block context would be needed to effectively implement such a ‘lazy relocation buffer’. For these reasons, we adopt the ‘eager’ approach.

4.1.2 Determining Contexts

As `bloop` processes a given instruction within a basic block, it must determine that instruction’s procedure context. Without inlining, the context never changes, since every instruction belongs to the same procedure. Consider now a procedure without loops, but with inlined non-loop code. In this case, as instructions are processed, adjacent instructions may belong to different alien contexts. Since we do not recover nested inlining, all of these alien contexts will be flattened with respect to the procedure scope, one for each alien file. An algorithm for this simple example is to first check whether the instruction’s source line information falls within the procedure’s bounds, and if not, to use a hash map to locate the appropriate alien context.

Now assume that the procedure has loops, inlined code, and possibly inlined loops. As was shown in Figure 3.8, the `bbranch-max-fuzzy` heuristic guesses a loop’s procedure context using the source line information from the loop’s backward branch. Assume for now that the guess is always correct and that the correct procedure context (possibly alien) is always chosen. As we process instructions in the correctly located loop, we may find inlined alien contexts within that loop. Since we want these alien contexts to remain in the loop (and not simply the procedure), we should flatten these contexts with respect to the loop and not the procedure.

Of course, it is often the case that `bbranch-max-fuzzy` guesses incorrectly. If this guess is incorrect, it usually results in a nesting cycle similar to that shown in Figure 3.9 on page 43 and described in Procedure Invariant 4. This means that the best procedure context is located not with the current loop, but in a shallower procedure

```

<P ...>
  <A1 ...>
    <L1 ...>
      <A2 ...>
        <L2 ...>
          ...
            <Am ...>
              <Lm ...>
                <Am+1 ...>
                  <s ...>
                    ...
  </P>

```

Scope s , embedded in a loop nest of depth m , has $m + 1$ alien contexts and one procedure context for a total of $m + 2$ procedure contexts.

Figure 4.1: Maximum procedure context nesting.

context (possibly alien). Since we wish to locate instructions ‘eagerly’ (as opposed to ‘lazily’), to be able to detect this nesting cycle, we need to check whether the instruction (or loop header) we are processing is best located in a shallower context, the current context, or an alien context associated with the enclosing loop. If the instruction should be located in a shallower context, then a nesting cycle has been found and must be corrected before continuing (Figure 3.10, page 44).

This algorithm is shown in Algorithm 4.2. Given an instruction and its corresponding line map information, it scans the current procedure contexts (*i.e.*, parents in the scope tree) and attempts to match one of them, giving preference to the procedure scope or a shallower alien scope. If a scope is found such that two procedure contexts form a nesting cycle, the cycle is broken using Algorithm 4.3. If no match is found, then existing alien contexts within the enclosing current loop or procedure are searched for a match; if this fails, a new alien context is created and associated with the enclosing loop or procedure. Note that because alien contexts are flattened with respect to loops, scanning the current procedure contexts is, for practical purposes, a constant time operation. In particular, for a loop nest of depth m , there can be

at most $m + 2$ parent contexts as illustrated in Figure 4.1. Even after inlining, loop nests rarely exceed a depth of 10.

Algorithm 4.2: determine-context

Input: Let scope s be a loop or statement whose context is unknown. Then s_e is s 's expected enclosing scope (loop or procedure) within Σ and σ its source line descriptor (from line map).

Result: The actual enclosing scope c (loop or procedure context) for s . As a side effect, Σ is modified to reflect c .

```

determine-context( $s_e, \sigma = \langle fnm, pnm, ln \rangle$ )
begin
  let  $c \leftarrow \text{NIL}$ 
  let  $c_e$  be the immediate procedure context of  $s_e$  (possibly  $s_e$ )
  let  $p$  be the enclosing procedure scope for  $s_e$ 

  // Search current contexts. Each  $c_x$  is alien except the last.
  foreach procedure context  $c_x$  in the path from  $s_e \rightsquigarrow p$  do
    if matches( $c_x, \sigma$ ) then
       $c \leftarrow c_x$ 
    end
  end

  if  $c \neq \text{NIL}$  then
    // Merge self-nested contexts
    if  $s_e$  is a loop and  $c$  is shallower than  $c_e$  then
      merge-broken-contexts( $s_e, c, \sigma$ )
    end

    // Ensure  $\sigma$  lives within loop (e.g., forward substitution)
    if  $s_e$  is a loop and  $\neg \text{matches}(s_e, \sigma)$  then
       $c \leftarrow \text{NIL}$ 
    end
  end

  if  $c = \text{NIL}$  then
    // Note that  $c_a$  is not shallower than  $c_e$ 
    let  $c_a$  be an alien context associated with  $s_e$  such that matches( $c_a, \sigma$ );
    or  $\text{NIL}$  otherwise
     $c \leftarrow c_a$  if non-NIL; otherwise, a new alien context initialized with  $\sigma$ 
  end
  return  $c$ 
end

```

Algorithm 4.3: merge-broken-contexts

Input: A loop scope s_f ('from') within Σ that should live within a different ancestor procedure context c_t ('to') because the statement scope represented by source line descriptor σ (from line map) lives both within s_f and c_t (representing a nesting cycle).

Result: Locate the loop nest represented by s_f within c_t .

merge-broken-contexts($s_f, c_t, \sigma = \langle fnm, pnm, ln \rangle$)

begin

let $x_2 \leftarrow s_f$

let $c \leftarrow \text{NIL}$

 // x_1 is the outermost loop nest containing s_f

let c and x_1 be ancestors of x_2 such that 1) c is the first ancestor procedure context of x_2 ; and 2) x_1 is a direct child of c on the path $x_2 \rightsquigarrow c$.
 (Possibly $x_1 = x_2$; a scope can be its own ancestor.)

 // Unnest the loop nest of s_f until it is a child of c_t . After

 // line 3 (above), if $c \neq c_t$ then we have:

//	$\langle P A \ c_t \rangle$		$\langle P A \ c_t \rangle$
//
//	$\langle A \ c \rangle$		$\langle A \ c \rangle$
//	$\langle L \ x_1 \ l="b_1-e_1">$	\Rightarrow	$\langle L \ x_1 \ l=" \sigma.ln - \sigma.ln ">$
//
//	$\langle L \ x_2 \ l="b_2-e_2">$	\Rightarrow	$\langle L \ x_2 \ l=" \sigma.ln - \sigma.ln ">$
//
//	$\langle L \ s_f \rangle$		$\langle L \ s_f \rangle$

while $c \neq c_t$ **do**

 Make x_1 a sibling of c

 // x_1 has just changed contexts. Make necessary fixes.

 // Note: c is an alien context

foreach scope z on the path $x_2 \rightsquigarrow x_1$ **do**

let $\langle b, e \rangle$ be the line bounds of z

$\langle b, e \rangle \leftarrow \langle \sigma.ln, \sigma.ln \rangle$

 Replicate c 's alien context for any child of z that cannot be contained within $\langle b, e \rangle$

end

$x_2 \leftarrow \text{parent of } x_1$

 Update c and x_1 according to line 3

end

end

Algorithm 4.4: matches

Input: A procedure, alien or loop scope s within Σ and a source line descriptor σ (from line map).
Result: Whether s could be a directly enclosing scope for σ .

matches($s, \sigma = \langle fnm, pnm, ln \rangle$)
begin
 // $\langle beg, end \rangle$: begin and end line matching tolerance.
 let $\epsilon \leftarrow \langle 0, 0 \rangle$
 switch *scope type of s* **do**
 case *procedure, p*
 $\epsilon \leftarrow \langle 0, \infty \rangle$ if p has a DWARF descriptor; otherwise, $\langle 2, 100 \rangle$
 if $\exists p'$, the next non-overlapping procedure to p **then**
 $\epsilon.end \leftarrow$ begin line for p' minus 1
 end
 case *alien*
 $\epsilon \leftarrow \langle \infty, \infty \rangle$
 case *loop*
 $\epsilon \leftarrow \langle 5, \infty \rangle$
 if s is within an alien context **then**
 $\epsilon.end \leftarrow 20$
 end
 end
 return *true iff the following hold:*
 1) if s is a procedure or alien scope
 a) s 's file name equals $\sigma.fnm$
 b) s 's procedure name matches $\sigma.pnm$ (allow for linkage characters)
 2) $\sigma.ln$ is contained in s given tolerance ϵ
end

4.2 GNU Binutils and Binary Analysis

bloop uses GNU Binutils 2.17 (the most recent release) to decode instructions and read debugging information. Because Binutils was designed to be used within the context of the GCC compiler and GDB debugger, it presents a number of design and algorithmic limitations for binary analysis [22]. We summarize the issues that most directly relate to **bloop**'s implementation.

The BFD library provides access to information for several different ABI's, but tends to have a 'lowest-common-denominator' interface, meaning *e.g.*, that BFD's interface to the binary's debugging information can be largely summed up with one routine that queries the line map.¹ While this design is sensible for a multi-target compiler back-end where the goal is abstract the details of many ABIs, it is not helpful for a binary analyzer. We therefore designed a 'thin' extension to the BFD interface to convey DWARF procedure descriptor information back to **bloop**.

Binutils contains decoders designed for disassembling a binary's instructions. However, the decoding routines are specifically designed for *printing* (since they are used by **objdump**), rather than for providing information to a consumer, one of which might be a printer. We therefore rely on a clever use of the print routine, implemented by Jason Eckhardt for the original **bloop**, to extract decoding information. Binutils' decoders have also required modifications to convey branch target addresses and instruction classification to a caller.

Besides interface problems, we also discovered algorithmic issues when applying Binutils to binary analysis. Specifically, BFD's line map query routine employed a linear search of the line map tables to find source line information for a given address. While this is acceptable overhead within a debugger, it becomes unmanageable when a linear search is performed for every instruction in a binary. We therefore modified several internal lookup algorithms to use binary search.

4.3 Lying Liars

Without debugging information **bloop** cannot recover useful results. It is also true that without *accurate* debugging information it cannot recover useful results. One

¹The routine is `bfd_find_nearest_line()`; it has a close cousin `bfd_find_line()`. Actually, Binutils 2.17 has added a routine to obtain some of the inlining information reported by GCC 4.x, `bfd_find_inliner_info()`.

of the challenges of performing accurate source code recovery has been gracefully managing debugging information that is either wrong or blatantly against DWARF specifications.

As an example of correcting wrong information, we found — much to our surprise — that the Intel 9.1 (x86-64) compiler generated wrong procedure address bounds on the AMRPoisson example driver of Chombo [32]. We began investigating when we noticed that *every* statement and loop in several procedures were wrongly attributed to the *same* line and source file. It turned out that the DWARF descriptor `addFabToSten(...)`² claimed that its object code ranged from `0x480102` to `0x4819e4`, or 6370 bytes. In fact, the procedure’s address bounds were from `0x480102` to `0x48022b`: a considerably smaller 297 bytes! This discrepancy caused Binutils — quite reasonably — to consult `addFabToSten`’s line map when looking up addresses between `0x48022b` and `0x4819e4`, and to return the last entry in the map. Since the only error of this kind that we have seen is an *overestimate*, we ‘solved’ the problem by ‘sanity checking’ the DWARF end address against information from the binary’s symbol table. Specifically, for a given procedure, we obtained the address of the next adjacent procedure within the symbol table (if non-existent, the end of the section) and set the procedure’s end address to the minimum of this address and the DWARF end address.

Besides pedestrian instances of erroneous information we have also had to handle the more exciting case of erroneous information that is also against the DWARF specification. While working with Trilinos [28] and the PGI 6.1-2 (x86-64) compiler, we (again) noticed that *every* statement and loop within most of the procedures were wrongly attributed to the *same* line and source file. Figure 4.2 illustrates the dilemma. Consider a query for address `0x40dba0`. Since DWARF requires the line

²`QuadCFStencil::addFabToSten(BaseFab<double> const&, DerivStencil&).`

PGI 6.1 on Trilinos 7.0.4			Intel 9.1 on Chroma 3.22.3		
Line map		Notes	Line map		Notes
<i>[Begin]</i>			<i>[Begin]</i>		
0x405b80	1		0x0	450	Bad address!
0x62f020	152	Data segment!	0x10	452	Bad address!
0x405b90	...		:		
:			0xe1	455	Bad address!
0x40dba0	226	Outside bounds!	0x4*aecac0	455	
:			<i>[End]</i>		
0x40d9b8	...				
<i>[End]</i>					

Figure 4.2: Erroneous DWARF line maps.

map to be sorted, the line map’s begin and end addresses specify its range.³ However, this address is located in the line map which begins with 0x405b80 and ends with 0x40d9b8. Hence, the out-of-order line map confused the query. To correct the out-of-order entries, we modified Binutils to sort the line map.

While the sorted the line map corrects the ordering problem, notice that the Trilinos line map of Figure 4.2 also contains an entry for address 0x62f020 — an address *located within the data segment*. Thus, correcting the invalid DWARF introduces a new problem: because of bad line map entries, the line map appears now to contain addresses between 0x405b80 and 0x62f020! Consequently, sorting the line map actually made the problem *worse* because it now appeared to overlap almost every other line map in the binary, creating ambiguity about which line map applied to a given address lookup. The problem was not an isolated incident. As Figure 4.2 also shows, the Intel 9.1 (Itanium) compiler generated a similarly erroneous line map for Chroma’s `hmc` [32]. While this line map was not out of order, it contained addresses such as 0x0 and 0x10 that were neither in the text section nor in the binary! Incredibly, several other line maps in the binary also contained entries for address 0x0. To

³*Cf.* §6.2.5 of [20, 44].

gracefully handle such data, not only did **bloop** need to sort the line map to obtain correct line map ranges, but it had to somehow ignore addresses not even in the text section.

We eventually solved this problem by patching Binutils' DWARF reader to filter obviously inappropriate addresses. To filter efficiently, we had to quickly find section bounds information and use this to determine whether to keep or reject a line map entry. Because of Binutils' implementation details, we chose to filter based on whether an address was within the `.text` section or not. In particular, we kept an address in the line map if and only if it was in the same region (relative to the `.text` section) as the first address of the sequence. These modifications enabled **bloop** to recover very accurate source code structure.

Chapter 5

Performance Visualization Using Dynamic And Static Structure

5.1 Deficiencies of Call Path Visualizations

HPCTOOLKIT's original visualization tool presented `csprof`'s call path profiles in a top-down 'file-browser' similar to Figure 5.3 on page 71. Each interior node in the visualization contained two pieces of information: a call site and the call site's host procedure, or its procedure frame. A node could be expanded or collapsed to reveal or hide its children. The tree could then be expanded in the style of a graphical file browser, with children sorted by inclusive metrics to enable quick zooming to the most costly call paths in the program. Using a mouse to click on a procedure frame caused the source code window to zoom to the start of the procedure's source code definition; clicking on a call site zoomed to the call site. Since call sites were not merged, distinct call sites formed distinct paths.

While this approach seemed promising, compiler transformations introduced problems. `csprof` collects call path profile data as a calling context tree [2]. In this representation, a path from the root to a leaf represents a call path where interior nodes are return addresses (the calling context) and the leaf is a sample point (instruction pointer). To convert the calling context tree into an effective and compelling visualization, `csprof`'s original correlation tool converted each return address into

a call site and then inferred its procedure frame. To convert a return address to a call site address, the tool subtracted either the width of an instruction packet (for fixed-width instructions) or one (for variable-width instructions).¹ To infer the call site's procedure frame, the tool queried the associated line map to obtain the call site's source line information. Assuming that the call site did not derive from inlined code, the tool created a procedure frame with the same name as the call site's source line information. It also computed a begin line for the procedure frame by using the source line for the first address in the associated line map. Neither of these methods worked well because of the reasons discussed in Section 3.1. Inlined call sites that caused omitted frames within a path were never marked as such. Moreover, a call site within inlined code appeared to execute from its original (non-inlined) location, while its frame's source file and line were mapped to whatever happened to be first in the line map. Both problems led to confusing visualizations. Another deficiency was that because the correlation tool had no knowledge of loops, it could not group call sites within a loop, thereby highlighting the costs incurred by the loop in context.

In effect, the problems with the visualization amounted to a lack of knowledge of the static source code structure. For example, if for a given routine the correlation tool could obtain nested inlining information, then the actual chain of missing frames could be recovered accurately. While our binary analysis techniques cannot actually recover a compiler's inlining decisions (*cf.* Section 3.3), they recover enough information to distinguish between code belonging to a native or alien frame and therefore result in enough information to inform the viewer that at least one static frame is missing due to inlining. Moreover, with accurate loop nests, call sites can be accurately attributed to the loop, even if it has undergone significant transformations.

¹Subtracting one from a variable sized instruction may result in an address *within* the call site instruction, but that is sufficient.

5.2 Combining Dynamic Call Paths with Static Structure

Given an object to source code structure mapping, combining the dynamic call path and static source code structure is a straightforward task. By making a copy of the run time load map that associates virtual address ranges with load modules, the call site and sample point addresses within the call path profile can be mapped to their appropriate load module.² Program structure information is computed for each procedure in the load module. Since each procedure is annotated with object address interval sets, the recovered procedure scope for the call site or sample point can be quickly obtained using a data structure such as a balanced tree. Within a procedure scope, statements form a fine-grained partition which implies that any given object address may belong to at most one statement and therefore at most alien context and one loop nest. Since each statement in the program structure is annotated with object-address interval sets, another lookup quickly identifies the associated source code statement, completing the static context of the call site: procedure, alien (if appropriate) and loop scopes.

Once the context of the call site has been found, the next step is to create a procedure frame for that call site and locate the call site within any loop nests. If the call site derives from an alien procedure context, then we can create two frames, one for the host procedure and one for the alien context, and locate both the alien frame and the call site within any enclosing loop nests. Recall that any potential nested inlining is flattened with respect the alien context's enclosing loop or procedure scope (*cf.* Section 4.1.2).

An additional benefit of the object to source code structure mapping is that object code disassemblies can be annotated with performance and program structure

²`csprof` captures the run time load map in its data file.

information. For example, all instructions associated with a loop can be easily observed. We have not actually implemented such functionality, but it would not be conceptually difficult to do so.

5.3 Case Studies

To demonstrate the effectiveness of combining dynamic calling context information with static source code structure, we present three case studies. We primarily compare HPCTOOLKIT against Tau, a tool based on instrumentation that is widely installed on large parallel machines and which supports both node-based and parallel analysis. Tau is quite mature, being the product of over fourteen years of joint development by the University of Oregon, the Research Centre Juelich and Los Alamos National Laboratory [39]. We based our experiments on Tau 2.16.2p2 configured with PDT 3.10 and procedure (`-PROFILE`) and call path instrumentation (`-PROFILECALLPATH`). We also configured a version with overhead compensation (`-COMPENSATE`).

For all case studies, we used a handicapped version of `csprof` that did not memoize call paths as described in [21] and which therefore incurred more overhead than it should have; nor did it collect information on the number of calls. `csprof` was run with the default sampling period of 1000 ms. Unless otherwise indicated, all our experiments were performed on an Itanium 2/Myrinet cluster running Linux 2.6.9 and using the Intel 9.1 compiler suite. Each compute node of the cluster is a dual-Itanium 2 running at 900 MHz with 4 GB of memory. When run times are reported, they are averages of three trials.

```

#include <stdlib.h> // for drand48
#include <map>
using namespace std;

class Mp : public map<int, double> {
public:
    Mp() { }
    virtual ~Mp() { }
    virtual void add(int i, double d) { insert(make_pair(i, d)); }
};

int main() {
    const int TenM = 10000000;
    Mp m;
    srand48(5);
    int ub = drand48() * TenM; // ~5.2M
    L1   for (int i = 1; i < 1000; ++i) {
        m.add(i, (double)i);
    }

    L2   for (int i = 1; i < ub; ++i) {
    S1       m.add(i + 1000, drand48());
    S2       m.add(i + TenM, drand48());
    }
}

```

delete

replace with: (double)i

replace with: (double)i

Source code for testing C++'s STL `map`. A second version of the code was formed by modifying the underlined text according to the boxes on the right.

Figure 5.1: Source code for testing C++'s STL `map`.

5.3.1 C++'s STL `map`

The first case study was based on the simple program in Figure 5.1 and was designed to test the effectiveness of compiler optimizations in hiding a simple object oriented abstraction. In this example, the `Mp` class is derived from the Standard Template Library's (STL) `map` class template and given a virtual member function `Mp::add`³ to wrap the insertion of elements. Far from being purely academic, the code represents a common and useful way of building C++ classes and of using STL and STL-influenced containers. In particular, STL's `map` is such a useful abstraction,

³While there is no direct need for `Mp::add` to be virtual, it is easy to imagine a larger example in which it this would be necessary. The destructor must be virtual.

that a developer might easily use it in place of a hash table, even though it is implemented using balanced trees and therefore does not provide the amortized time bound guarantees that are typical of a hash table. This example uses `int` keys rather than strings to represent the more reasonable decision of using `map` with a pointer-valued key.

The code in Figure 5.1 inserts approximately 10.4 million items into a STL `map` over the course of loops L_1 and L_2 . A quick scan of the `main` routine suggests that the bulk of the execution time should be consumed by these loops, with the latter loop dominating. Even though `Mp` has virtual functions, an optimizing compiler could inline the calls to `add` and the `map<>::insert` routine. Loop L_1 could be completely unrolled (*i.e.*, eliminating the loop), though this seems unlikely and unnecessary.

We created three versions of the program, one with the Intel compiler and two with the Tau compiler, with and without overhead compensation. All three versions were compiled with `-O3 -g` and were run on a single dedicated node. When executed under `csprof`, the run time increased from 41.46 to 42.92 seconds (or 3.5% overhead). In contrast, the Tau version without compensation ran in 153.07 seconds (270% overhead) while the version with online compensation ran in 179.48 seconds (333% overhead). To obtain full calling contexts, we set the Tau call path depth to a sufficiently large number (1000).

We then proceeded to examine Tau's process-based displays (not shown). Results for the two Tau versions were not significantly different. The Call Graph display, a graph visualizer with nodes and edges, showed that `main` called `Mp::add` as well as the `Mp` constructor and destructor, but had no information about `drand48` or any STL routine. Already we see that in the Tau version, `add` was not inlined as expected; nor did Tau detect the call to `srand48` or the calls to `drand48`. The Statistics Table, the view we judged to be the most useful, showed that `main` called `add` a little over

10.4 million times and that 90% of the inclusive time was spent in `add`. All told, the information is trivial except for the relative times, and the overhead percentage makes even this suspect.

To improve Tau’s default output, we tried two things. First, Tau’s overhead can be reduced by manually configuring a ‘throttle’ mechanism to selectively instrument only certain routines.⁴ Since the default throttle rule is to disable profiling of a function that is called greater than 100,000 times and which has an inclusive time of 10 microseconds, it would certainly apply to `add`. Unfortunately, `add` (along with its callees) is just the routine that we want information about! For throttling to be useful in this case, Tau would need to somehow inform the analyst when `add` was throttled and how that time compared with the total execution time; in other words, to expose the ratio of non-throttled to throttled time. Note that sampling based profilers are based on just this sort of scaling by assuming that a large enough collection of samples represents the whole. Nevertheless, we tried the ‘throttle’ feature using a rule to disable profiling after 1 million calls, about 10% of the 10.4 million identical calls to `Mp:add` (not shown). This change reduced overhead to 9.8% (45.54 seconds), but it also inverted the relative exclusive times of `main` and `Mp:add` to where `main` was about 15 times more expensive than `Mp:add`! This places the analyst in the awkward position of guessing how many calls to the key routine can be ignored while still generating reasonably accurate results.

The second thing we tried was adding Tau’s loop instrumentation. Because of the potential for overhead explosion, Tau does not perform loop instrumentation automatically and, even then, only outer loops can be instrumented. Moreover, the user

⁴‘Throttling’ is accomplished by manually configuring an environment variable that contains a rule for determining when to stop collecting data for a function. The process is manual because if the default rule is inappropriate, a new one must be developed without the aid of an automatic feedback loop.

Name	Inclusive Time %	Exclusive Time %	Calls	Child Calls
int main() [(t1.cpp) {12,1}-(26,1)]	100.0%	2.0%	1	4
Loop: int main() [(t1.cpp) {22,3}-(25,3)]	98.0%	7.8%	1	10,496,788
void Mp::add(int, double) [(t1.cpp) {9,3}-(9,64)]	90.2%	90.2%	10,496,788	0
Loop: int main() [(t1.cpp) {18,3}-(20,3)]	0.0%	0.0%	1	999
void Mp::add(int, double) [(t1.cpp) {9,3}-(9,64)]	0.0%	0.0%	999	0
void Mp::Mp() [(t1.cpp) {7,3}-(7,10)]	0.0%	0.0%	1	0
void Mp::~Mp() [(t1.cpp) {8,3}-(8,19)]	0.0%	0.0%	1	0

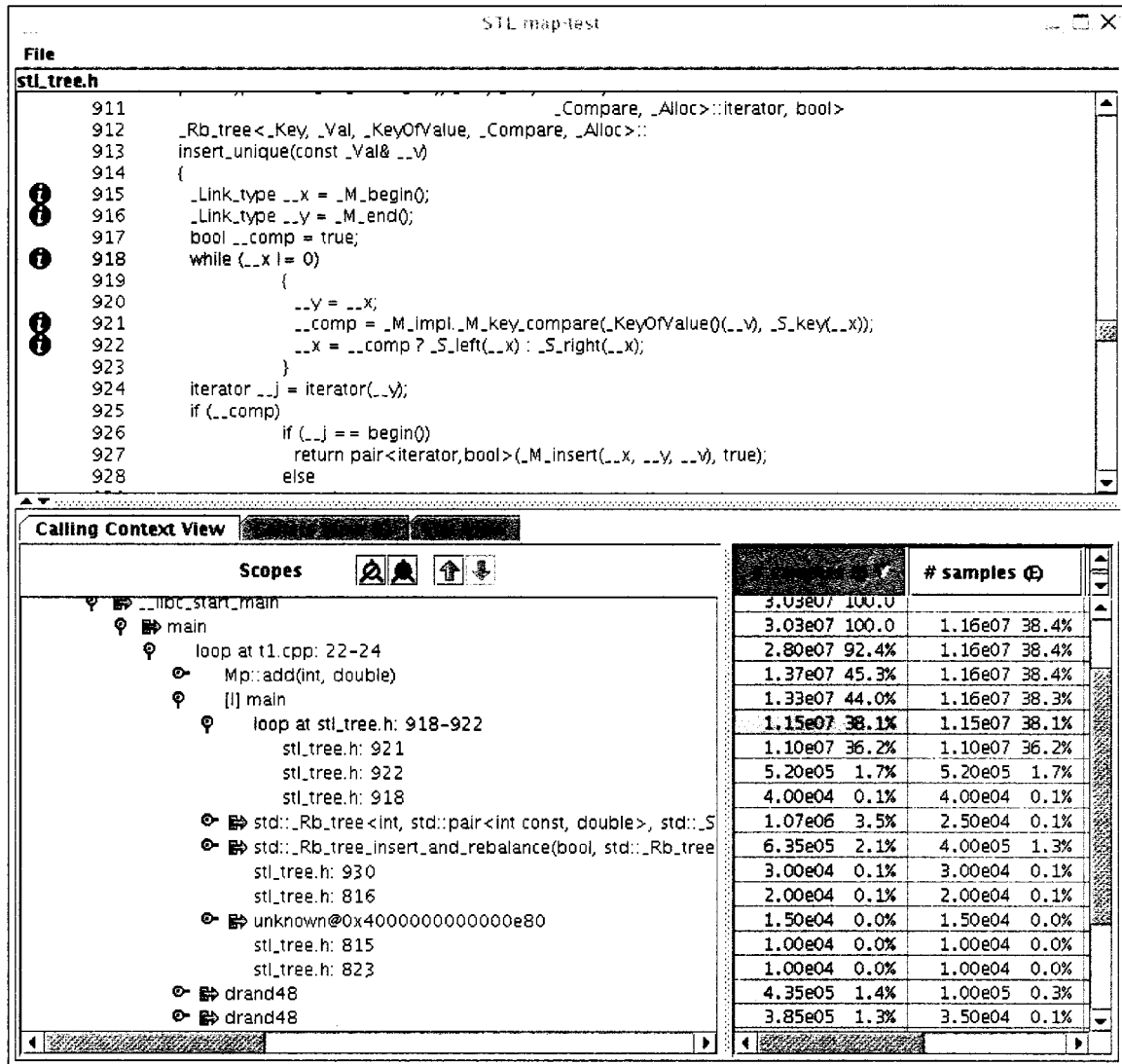
Figure 5.2: Tau’s visualization of STL map example (Figure 5.1).

must manually create a loop instrumentation file with the appropriate instructions.⁵ While this is impractical for large codes, for this small example, it was easy enough to add appropriate instrumentation. The result, shown in Figure 5.2, was more informative than the first, indicating that the second loop consumed 98% of the execution time, but much about the performance remained unclear.

The most salient unknowns were details of the STL map implementation. Why did we not see any STL calls? Further investigation revealed that Tau does not instrument header files — in which STL is implemented — though it is possible to manually instrument a particular header file and then modify the order of the compiler’s include path lookup to find the instrumented version first. Since our purpose was to evaluate performance *tools*, we did not think such manual heroic efforts were warranted.

When we used HPCTOOLKIT to visualize the profiling data, several things immediately became apparent, some of which are shown in Figure 5.3. According to this data, loop L_2 consumed 92.4% of the execution time (similar to Tau’s estimate), but the remaining 7.6% of the time derived from an inlined Mp destructor! In contrast, Tau assigned negligible times to both the Mp constructor and destructor. Furthermore, HPCTOOLKIT clearly distinguished between the two loops in `main` and also

⁵This can be created automatically on a per file basis with Tau’s Eclipse plug-in.



A loop within the STL `map<>::insert` routine that consumes, within this context, 38.1% of the inclusive time. The loop's "[I] main" parent indicates that the loop was inlined into `main`. Closer inspection with HPCTOOLKIT reveals that this loop was inlined into loop L_2 from the call to `Mp::add` at S_1 (nested inlining).

Figure 5.3: HPCTOOLKIT's visualization of STL map example (Figure 5.1).

differentiated the loops themselves from the other call sites in `main`, such as the inlined destructor.

Expanding loop L_2 revealed some fascinating details. Most significantly, unlike Tau's data, this two-line loop contained several callees, some embedded within a loop.

Two of the children consumed large fractions of the execution time and therefore would have been quickly noticed, but the others did not and therefore would mostly likely have been ignored had they not been grouped into the loop's cost. For example, the two calls to `drand48`, distinguished by their call sites at lines S_1 and S_2 , respectively, each consumed 1.3% of the execution time; one could even follow the call path further to learn that `drand48` is implemented with `erand48_r`. Loop overhead appeared to be minimal, with only a few samples attributed to code directly inside the loop. By far, most of the inclusive time (89.6%) was attributed to two call sites: a call to `Mp::add` deriving from the call site at S_2 (45.3%) and an inlined STL `map` insert routine deriving from S_1 (44.3%)! In other words, one call to `add` was inlined into L_2 (along with STL routines) while the other was not! Moreover, STL routines from the inlined `add` at S_1 had been inlined into L_2 (nested inlining). Expanding the `add` call site (at S_2) indicated that the same STL `map` insert routine that had been inlined directly into L_2 (from S_1) had also been inlined within `add`. While this asymmetric inlining seemed surprising, we verified the diagnosis against both bloop's program structure data and a disassembly of the object code. Given the amount of STL inlined into the loop, a possible explanation for this seemingly bizarre compiler decision is that a heuristic designed to limit the amount of code inlined into a loop prohibited inlining of the second call to `add` at S_2 .

A surprising number of STL implementation details can be gleaned in a very short amount of time from this data. An analyst moderately familiar with data structures, but not familiar with STL implementations, would be able to quickly 'guesstimate' several details by the names of the container types and routines. For example, the inlined insert routine is named `Rb_tree::insert_unique`, suggesting that STL maps are implemented by Red-Black balanced trees and that keys are assumed to be unique. Internal STL call sites such as `Rb_tree.insert_and_rebalance`

As a third point of comparison, we tried Apple’s Shark tool, one of the best process-based performance tools we are aware of from either vendors or researchers. Shark contains a sampling-based call path profiler which we applied to a version of the code compiled with GCC 4.x (using `-O3 -g`) on Mac OS; a screen shot is shown in Figure 5.4. Shark was able to measure the optimized code, though because it merges all call sites to the same callee from the same caller, it was not able to provide as much context as HPCTOOLKIT. For example, Shark did not distinguish between distinct calls to the STL `map` insert routine. More importantly, it does not show the effects of inlining or aggregate costs to loops. A surprising aspect of this data is that nearly 65% of the inclusive time was spent in `map<>::erase`, supposedly called directly from `main`. While this sample code is small enough that we know that the call derived from a static destructor in `main`, this fact is not exposed because the destructor’s frame is missing in the call path. Moreover, loops from the insert routine are not exposed. However, unlike HPCTOOLKIT, Shark does have a nice object code to source code correlation tool, though it is based only on source lines and not program structure.

In summary, our unique approach of merging call path profile data with program structure information, reveals essential information about compiler optimizations and implementation details — all correlated with profiling data — that would not otherwise be available. To emphasize once again the utility of merging static program structure with call path data, we slightly modified the underlined code of Figure 5.1 according to the boxes in the corresponding right margin. Surmising that the use of `drand48` prevented some loop optimizations, we removed or replaced references to it. The resulting loops are candidates for fusion if the compiler has knowledge that the order of map insertions is irrelevant. Indeed, HPCTOOLKIT’s resulting visualization showed that the loops *had* been fused. In contrast, Shark could not present loop effects and Tau’s loop instrumentation *prevented* the fusion. Although it is true that

the fused loop identified by `bloop` did not directly correspond to source code, this is the sort of incongruity that a performance analyst desires. As one final example, after correlating `bloop` information with a flat profile of the Navier-Stokes solver S3D (Sandia 3D Direct Numerical Solver) [12], we immediately noticed a small loop consuming 5% of the L1 cache misses (collected using HPCTOOLKIT’s flat profiler) — a loop for which there was no corresponding source loop and which mapped to a call site. After briefly wondering if we had found a bug, we noticed that the call used Fortran array notation to pass a non-contiguous 4-dimensional slab of a 5-dimensional array to the callee. Because the callee was unprepared to accept this non-contiguous 4-dimensional slice, the compiler had made a *copy* of the slice into a stride-1 array and passed that, giving rise to a loop that incurred a relatively significant performance loss.

5.3.2 Chroma’s hmc

As a second case study we investigated the lattice quantum chromodynamics solver Chroma [31]. Chroma is a very large (with binaries of approximately 110 megabytes), modular application designed around C++ expression templates. Its modular design means that understanding costs in context is very important. Because of its expression templates, at compile time myriads of very complex templates are instantiated and possibly inlined. Consequently, Chroma can take hours to compile: about four on a 1.3 GHz Itanium/Linux node.

This fact exposes one of most significant practical differences between HPCTOOLKIT’s approach and Tau’s. While `bloop` requires that applications be compiled with debugging information (*i.e.*, `-g`), such information has no important effect on performance and no effect on compiler optimizations; it is therefore hardly a burden in either production or development environments. In contrast, instrumenting Chroma

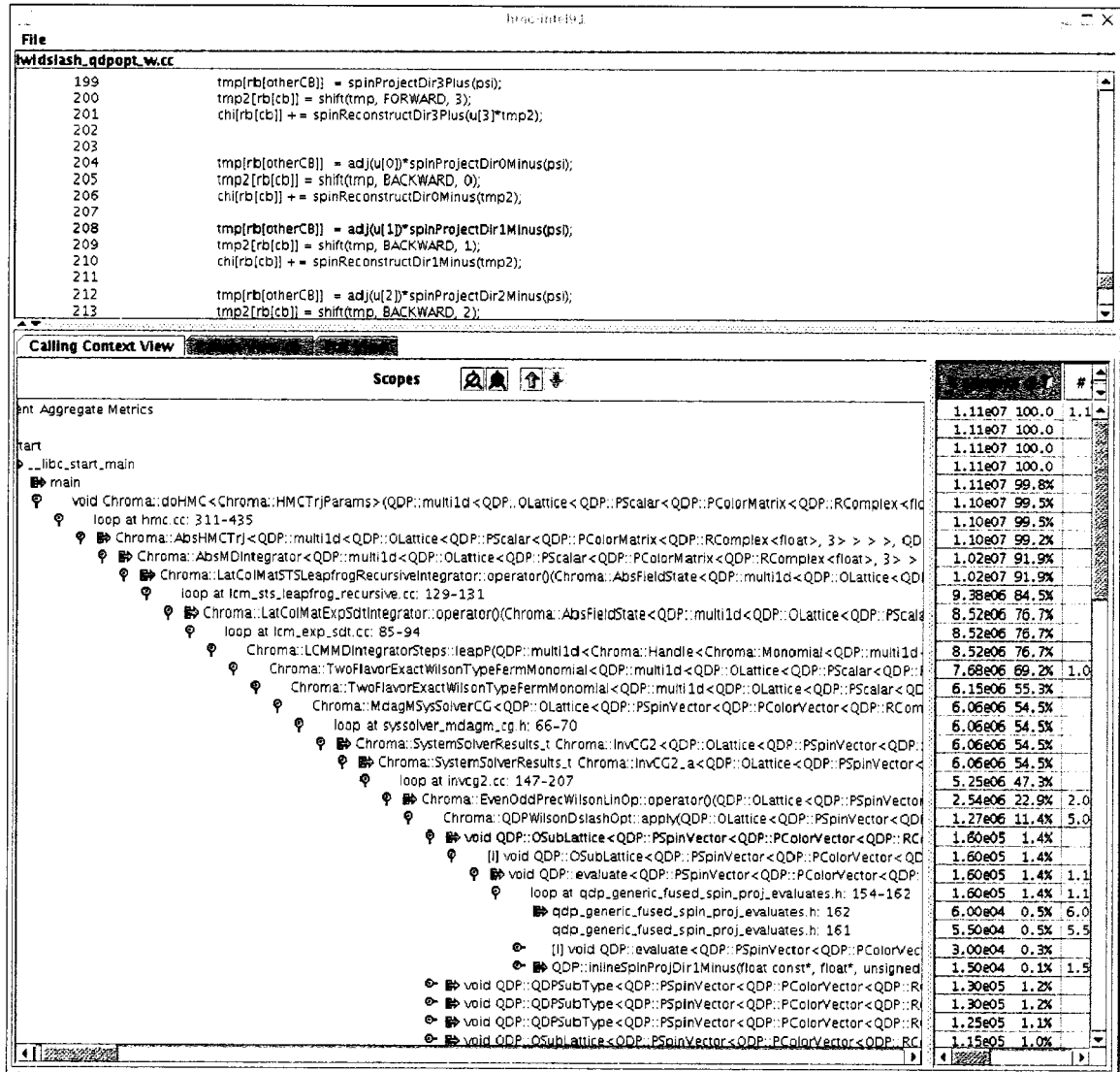
required a complete recompilation of the entire Chroma application, along with its libraries QDP++ and QMP. The time for recompilation with instrumentation increased from four hours to about seven. While the overhead of static instrumentation is often unacceptable in production environments, this sort of recompilation overhead is likely to be similarly inconvenient for general development. At the very least, such times are a large impediment for all but the most foresighted of analysts.

We ran Chroma’s `hmc` driver on the provided sample input file.⁶ Tau’s overhead became more manageable on a code with non-trivial functions, dilating the base execution time of 16.39 seconds by 35% to 22.25 seconds. However, when we increased the call path depth limit from 2 to 3000, collecting full call paths increased the time dilation to 123% (36.56 seconds). In contrast, `csprof` collected full call paths (without memoization enabled) for a dilation of 2.14%. We hypothesized that Tau’s online overhead compensation might perform more favorably on this code than the STL map code (Figure 5.1). However, we were unable to collect data because of a run time error. Sample screen shots for Tau and HPCTOOLKIT are shown in Figures 5.5 and 5.6, respectively.

The contrasting dilation effects were apparent even at the top level. HPCTOOLKIT attributed 99.5% of the execution time to the `doHMC`⁷ driver routine while Tau assigned only 88.2%. Another difference became immediately apparent while expanding the most expensive paths according to each tool: Tau appeared to have missing frames. Specifically, the `doHMC` routine applied arguments to a templated `AbsHMCTrj` functor object to yield a call to its virtual `operator()` routine. HPCTOOLKIT identified both the call site — nested within the main loop on lines 311-435 — and callee. Because the compiler created a concrete instance of the `operator()`

⁶The input file was `hmc.prec.wilson.ini.xml`.

⁷Within this section, all classes and routines are assumed to be in the `Chroma` name space unless otherwise indicated.



The 'hot' path to the solver is expanded, embedded within five dynamically nested loops. The highlights show a call site for an `operator=` expression template. Deeper within the call path are some inlined routines and loops implementing the `operator=`. (The function name is not visible because of the type qualifiers!) Observe that siblings to this call site consume about 1% of the inclusive time making it difficult to know where to focus.

Figure 5.6: HPCToolkit's visualization of Chroma (calling context tree).

dynamic invocations of `dsdq` and `getX`⁸ as well as an application of a `MdagMSysSolverCG` object, significantly reducing important context. Because such dynamically invoked routines are characteristic of object-oriented design rather than the exception, any effective tool must account for them.

Following the most expensive path down to the main solver interface `InvCG2` showed that the attribution discrepancies had grown. While `HPCTOOLKIT` charged 54.5% of the execution time to this path, Tau assigned only 32.8%. `HPCTOOLKIT` also showed how the solver’s interface called a worker routine that contained the solver’s main computational loop (47.3%). This worker routine called about 13 routines, but the loop path — about 87% of the solver’s 54.5% — clearly focused our attention on the 7 call sites *within* the loop. In particular, two distinct applications of `EvenOddPrecWilsonLinOp` objects consumed 22.9% and 22.8%, respectively, of the execution time — call sites that Tau represented with one frame. `HPCTOOLKIT` attributed almost all of the time consumed by one of the `EvenOddPrecWilsonLinOp` object applications to two other distinct applications of `QDPWilsonDslashOpt` objects, each consuming about 11.5% of the inclusive time (about 46% total). In contrast, Tau combined all four call sites together for a cost of 30.2%.

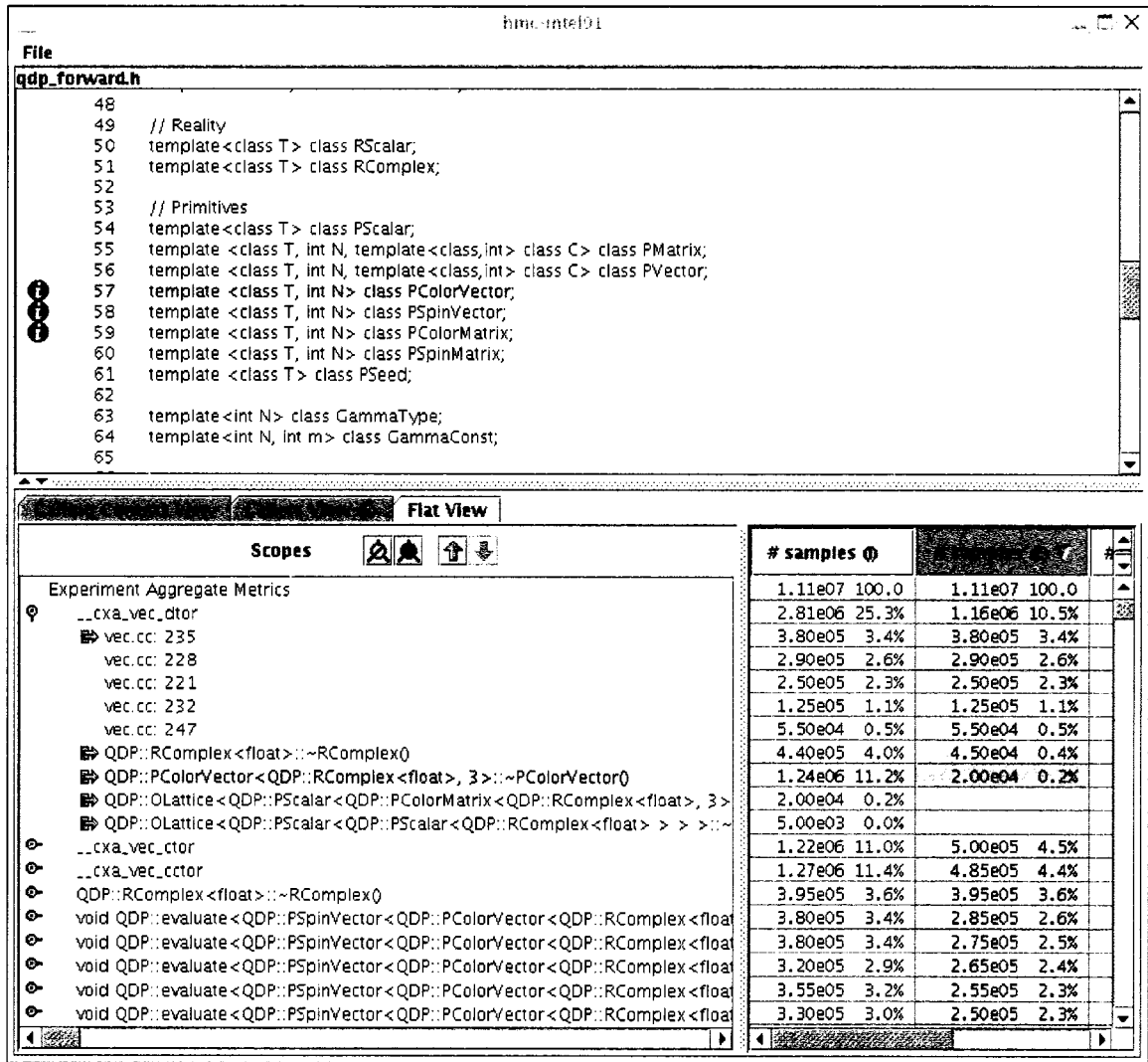
When expanded fully, Tau reported one additional frame on the hot path, a call to an allocator function consuming 3% of the time, for a total depth of 9 frames of context (prior to the allocator call). In contrast, `HPCTOOLKIT` presented the full object code context of 14 procedure frames (excluding three frames of context for `main!`) and five loops. Moreover, there were several actual frames that Tau elides between a `QDPWilsonDslashOpt` object application and the allocator function. In effect, by not instrumenting header files, Tau missed all of the internal context of the solver.

⁸Both are member functions of the `TwoFlavorExactWilsonTypeFermMonomial` class.

HPCTOOLKIT helped elucidate the complex effects of expression template coding. Expanding calls made by an application of the `QDPWilsonDslashOpt` object revealed about 30 distinct call sites, mostly to expression template operators with fully specialized names that would make even the most efficient hash function cringe. Most of the call sites are responsible for approximately 1% or less of the of parent’s 11.6% inclusive time. Expanding the most expensive one (1.4%) revealed a call to an `operator=` function tagged with `inline`, but that had *not* been inlined. Rather, this operator itself contained inlined code along with a calling path containing several additional frames and a loop, responsible for the 1.5%. While this information was interesting, it was difficult to determine where to go next because the costs were so widely distributed. It is likely that group information would have been helpful.

This shows that when costs are finely distributed across contexts, calling context views do have limitations. In this case we could not easily distinguish between the individual templated expression operators to determine if there was a clear performance problem. Therefore, we turned to HPCTOOLKIT’s ‘flat’ view, shown in Figure 5.7. Conceptually, this view ‘flattens’ the calling context of each sample point and then combines context-less samples from the same procedure (maintaining the static structure information) to generate exclusive times. However, because the view is computed from the merged calling context tree and program structure, it also computes *inclusive* times (unlike a IP-histogram) and exposes call sites (including inlined ones) and loops. The flat view showed that nearly 50% of the execution time had been spent in class constructors and destructors! This fact shows that C++ class abstractions can have a high price.

In summary, we have seen that HPCTOOLKIT enabled us to begin to understand both the operation and performance of a very complex code that depends on compile time transformations and dynamic function resolution. We were able to relatively



A flattened view, sorted by exclusive time. It is immediately apparent that the procedures `__cxa_vec_{dtor,ctor,ctor}` and their descendants consume approximately 47% of the total execution time! The highlights show the instantiation point (*i.e.*, a ‘definition’ created at compile time) of a `PColorVector` destructor.

Figure 5.7: HPCTOOLKIT’s visualization of Chroma (flat).

quickly delve into the computational kernel; at one point, loop information quickly filtered between interesting and uninteresting call sites. However, this example also showed that flattening calling context information in the presence of many very short procedure calls can be valuable.

5.3.3 NAS Parallel Benchmark’s CG (UPC Version)

As a third example, we present data for a Unified Parallel C (UPC) [19] version of the NAS 2.3 Parallel Benchmark CG. UPC is an extension of C that provides a global shared address space (partitioned) for Single Program Multiple Data (SPMD) programs. By supporting C’s flexible data structures, UPC enables programmers to write parallel applications involving unstructured data and port them between both shared memory and cluster architectures. The CG benchmark computes the conjugate gradient of a large, sparse, symmetric positive definite matrix. The kernel is typical of unstructured grid computations in that it tests irregular long-distance communication and employs sparse matrix-vector multiplication [43]. It is therefore a good application for UPC.

For this study we used a UPC version of CG written by Cantonnet and El-Ghazawi [11] and modified (via loop unrolling) by Coarfa, Dotsenko *et al.* [15]. We used the Berkeley UPC compiler [13] (version 2.4.0), configured with the Intel 9.0 compiler on the same Itanium architecture. We collected `csprof` data for a class B problem size on four nodes and examined the results of one node.

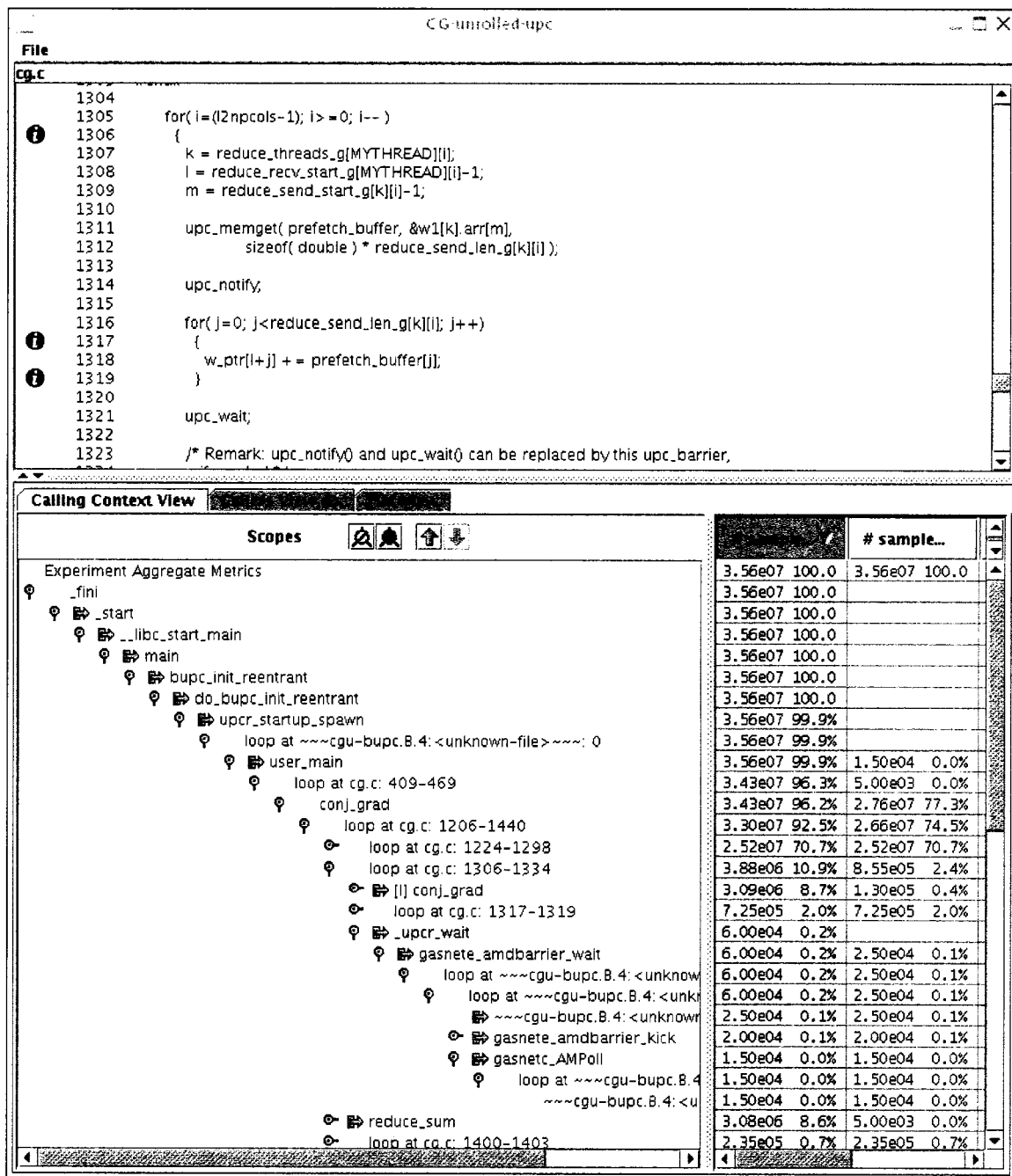
We did not include Tau results because Tau does not yet include a configuration for UPC or for UPC’s communication library GASNet [8]. Moreover, while it would be possible to instrument the main CG solver code manually, because the interesting portions of the call graph are the communication calls induced by non-local memory references, the Tau data would be trivial. This reinforces one of the benefits of our binary analysis method: as long as a compiler provides basic debugging information, intriguing details can be obtained without either obtaining source code or rewriting compilation scripts to manually instrument source files. Berkeley’s UPC is therefore interesting not simply because UPC has important applications, but because it is not a tightly coupled vendor compiler. Rather, it is a source-to-source compiler that

generates C source code (containing calls to run time library routines) that is then translated to object code by a vendor compiler.

HPCTOOLKIT's visualization, a screen shot of which is depicted in Figure 5.8, immediately shows the Berkeley UPC run time layers that launch and initialize inter-process communication. While this overhead was negligible for a 51 second run, it is clear that the initialization process was non-trivial. For example, the HPCTOOLKIT visualization showed that before CG's main routine was invoked, there was a call to the Berkeley UPC run time initialization function `upcr_startup_init` which descended through thirteen call sites and three loops to initialize three communication layers: GASNet, MPI (only used as a launcher), and the low-level Myrinet interconnect driver. All this was observed even though neither the specific MPI installation's nor the Myrinet driver's source code was available.

Expanding the hot path, we quickly found the most important computational loop within `main`, despite the fact that it is a 350 line routine with many other loops. This loop consumed 96.2% of the execution time and called the conjugate gradient solver, `conj_grad` which consumed 92.5% of the time. The rest of the routine's execution time was mostly consumed by computation of the residual norm (loops) and calls to `reduce_sum`. The static data clearly distinguished the key loop from other ancillary loops and procedure calls.

The main `conj_grad` loop (92.5%) is itself comprised of several inner loops and calls to `reduce_sum`. This main loop contains a `upc_barrier` statement, which, interestingly, we could easily see was implemented by calls to `_upcr_wait` and `_upcr_notify`. We could also expand these call chains and descend into both the GASNet and Myrinet communication layers. However, the loop by far spent most of its time in three areas: two doubly nested loops and a call to `reduce_sum`. The first doubly nested loop (70.7%) contains the computational core (and the loop that was mod-



The 'hot' path to the main `conj_grad` loop (1206-1440) is expanded. Within this loop are two loops (1224-1298 and 1306-1334) and a call to `reduce_sum`. The second loop (1306-1334) is expanded. Highlighted is a call site to `_upcr_wait` mapping back to a `upc_wait` statement within this loop. The path that `_upcr_wait` takes down into the GASNet layer is also shown.

Figure 5.8: HPCToolkit's visualization of NPB 2.3 CG.

ified by unrolling). By simply glancing at the loops nested in the first loop, we could see that the compiler had inserted no communication calls and that the loop's performance depended on effective use of the memory hierarchy and the processor's functional units. The second doubly nested loop (10.9%) *did* perform communication: a call to `upc_memget` was replaced with an inlined call to `_gasnet_get_bulk` that consumed 6.1% of the inclusive time. A `upc_wait` statement, translated to a call to `_upcr_wait`, was negligible, at least on four processors. The call to `reduce_sum` (8.6%), not surprisingly, used communication to perform a reduction. Actually, there is another call to `reduce_sum` within the loop — and as previously noted outside of the loop — but the distinct contexts enabled us to clearly see that this particular one was by far the most expensive. Within `reduce_sum` there was an inlined call to `_gasnet_put` that corresponded to a simple C assignment. Because the assignment occurred when the loop induction variable `rs_i` was not equal to `MYTHREAD`, it referenced remote data and induced non-local communication.

This case study is illustrative in several ways. First, it shows that even on a relatively small benchmark, correlating dynamic call path information with static program structure enables the analyst to quickly focus on the most costly loops of the program as well discover interesting facts about a compiler's inlining decisions. In every case, the loop bounds that `bloop` computed were exact and showed how the computation was structured. Moreover, they enabled us to quickly divide between loops that computed on purely local data and loops that induced communication. Second, because our methods do not rely on source code, program structure information was available even when source code was not: from procedure names, to loops, to alien procedure contexts. We observed that what might remain a simple C assignment on a shared memory architecture, translated into a communication call — and we traversed the two layers of communication libraries that implemented it (GASNet

and Myrinet). Third, our methods were able to provide all this information in the context of a research compiler thinly layered on top of a vendor compiler. While it is true that collecting good structure information was only possible because of the Berkeley compiler's inclusion of `#line` tags to facilitate debugging, it highlights that as long as a binary has basic debugging information, our methods provide extremely useful interpretive insight where other tools fail.

Chapter 6

Conclusions

We have designed a method of binary analysis for effective attribution and interpretation of performance measurements on fully-optimized code. Specifically, 1) we have rewritten HPCTOOLKIT's binary analysis tool `bloop` to compute an object to source code structure mapping for fully optimized binaries; 2) We have extended HPCTOOLKIT's correlation tool to combine call path profiles with source code structure by exposing inlined frames and loop nests as cost-inducing entities; and 3) We have presented visualizations of the correlated data and used them to analyze complex codes.

Of course, correlating dynamic and static information is not difficult for source code that closely resembles the target architecture and which is not transformed by a compiler. However as scientific applications are designed around object-oriented abstractions, they rely more on compiler optimizations. Our binary analysis methods recover the source code structure of the most important components: procedures and loops. We have presented ideas for creating class member function groupings and have observed that additional information is necessary to recover inlined nesting structures.

In one sense, we have solved an artificial problem. In 1992, Brooks, Hansen and Simmons [9] developed debugging extensions, generated by the Convex compiler, that mapped object code to a scope tree of (Fortran and C) routines, loops, blocks,

statements and expressions. While they left to future work a solution for the inlining problem, if compilers had adopted the revisions they described, both debuggers and performance tools could be more revealing. Nevertheless, however easy the problem of creating the object to source code mapping could have been, the fact remains that the most aggressive compilers used for scientific computing generate limited debugging information. Although it is true that compiler writers must weigh market considerations when allocating developer resources between new features, bug fixes, testing, and documentation, a relatively small amount of effort by compiler developers would yield much low hanging fruit:

- Simple exploitation of the current DWARF standard would enable powerful techniques for understanding performance. For example, two simple changes would enable detection of all inlined code (with the exception of the last function in a procedure-scoped class; *cf.* Section 3.2.3). First, while DWARF does not represent procedure end lines, it would be trivial to generate at least a partial DWARF descriptor for *every* source code procedure and template instantiation, whether inlined at every call site or not. Second, procedure-scoped class descriptors should be nested within their containing procedure. Additionally, recording inlining decisions — which have been representable in DWARF for several years — would make associating inlined code with call sites trivial.
- Compilers should generate *correct* DWARF. None of the invalid DWARF we have encountered could be justified by citing a DWARF design flaw or representational deficiency.
- The major source code construct not representable in DWARF is a loop. One very simple piece of compiler support for loop recovery would be to always map a (structured) loop's backward branch *not* to the condition test, but to the

loop’s header (because the condition test itself may derive from inlined code).¹ However, even this would not enable an analyzer to account for transformations such as loop interchange found in automatically parallelizing compilers. Because loops are important, we advocate DWARF extensions that at least describe source loop nesting structures and associate them with address ranges. Even in this context, object code analysis could still be used to detect compiler generated loops and loops formed with unstructured control flow.

- Line maps should have source *column*-level granularity, enabling the tracking of source code expressions. This would provide a simple way to account for macro function expansion by extending the `#line` directive to set the column as well as source line and file as shown in the following example:

```

        z = foo(
#line macro-begin-line macro-begin-column macro-file
               expanded-macro-source-code
#line restored-line restored-column restored-file
        )

```

While all these proposals result in larger amounts of debugging information, this is a small price to pay in the context of scientific computing. The extra space requirements should not affect performance since debugging sections need not be loaded for production runs. If space is an issue, compression techniques can appropriately trade time for space.

Although we have focused on common languages for scientific computing such as C, C++, and Fortran, our principles have broad application for any statically compiled language where performance is critical. Because we have based our analysis methods only on a ‘lowest common denominator’ set of DWARF debugging informa-

¹Current architectures use separate instructions (or operations, in the case of VLIW architectures) for evaluating the loop continuation condition and branch.

tion and have tested them on highly optimized binaries with compilers that aggressively transform both procedures and loops, they should be easily adaptable to other languages. While our loop recovery is based on the `bbranch-max-fuzzy` heuristic that does make an assumption about typical compiler behavior to identify forward substitution, the main assumption behind it is machine-independent (loop invariant code motion) and therefore unlikely to be invalidated by a future compiler optimization. We have also seen that in the worst case, loop heuristics can make errors that affect only the procedure whose structure is being reconstructed.

When compared with instrumentation-based techniques, our measurement and analysis methods have several advantages. First, sampling-based call path profilers introduce minimal distortion during profiling and allow arbitrary compiler transformations; on many operating systems they can even be invoked on unmodified binaries. In contrast, static instrumentation introduces side effect inducing code into an application, significantly dilating execution time and preventing important procedure and loop optimizations. We have found that on large codes, recompiling to add instrumentation is at best painful and at worst infeasible.

Second, using binary analysis to recover source code structure is uniquely complementary to sampling-based profiling. While Tau [39] only instruments non-header source files, HPCTOOLKIT's call path profiler samples the whole calling context irrespective of whether the call chain passes through communication libraries or process launchers. Moreover, `bloop` recovers the source code structure for any portion of the calling context regardless of source code (as long as debugging information is present). These facts are significant because even if we could obtain a perfect object to source code mapping from a compiler, an absence of source code surely implies (in practice) the absence of such a mapping. In other words, using binary analysis to recover source code structure handles the complexities of real systems.

Third, binary analysis is an effective means of recovering the source code structure of fully optimized binaries. When source code is available, we have seen that **bloop**'s object to source code structure mapping accurately correlates highly optimized binaries with procedures and loops. Among other things, it accounts for inlined routines, inlined loops, fused loops, and compiler generated loops. In effect, our binary analysis methods have enabled us to observe both what the compiler did and did *not* do to improve performance.

We conclude that combining call path data with static source code structure provides unique insight into the performance of modular applications that have been subjected to complex compiler transformations. **bloop**'s program structure information for the STL `map` example (Figure 5.1, page 67) transformed what might have been trivial or confusing call path information into a series of interesting details. We saw unexpected compiler inlining decisions and surprising C++ destructor overheads. We quickly learned details about `map`'s implementation (Red-black trees, location of the main insertion loop) that might have been more difficult to find by simply reading the source code! Our visualizations of CG enabled us to quickly find key loops in the solver and distinguish between local and non-local computations. We identified inlined communication and observed how the Berkeley UPC compiler [13] converted non-local references to communication. Our analysis of Chroma [31] showed **bloop**'s ability to handle very large and complex codes designed to undergo complex compiler transformations. Our methods recovered the structure of a deep call path through several dynamically dispatched routines, including a deep, dynamically nested loop.

Yet, Chroma also showed the limitations of typical top-down calling context visualizations. Once we had expanded the 'hot' path and found the main computational loop, costs were so finely distributed among expression template operator instantiations that it was difficult to understand if there was a performance problem.

By exploiting HPCTOOLKIT’s ability to flatten the calling context tree, we quickly observed that an extremely large percentage of the execution time was devoted to managing C++ object deletion.

It should be said that Tau’s research group — Maloney, Shende, *et al.* — has worked on eliminating some of the problematic effects of static source code instrumentation that we have observed. To support instrumentation of fully optimized code, Shende developed an ‘instrumentation aware’ compiler [38]. This compiler consumed instrumented code, but removed the instrumentation before performing analysis and optimization. Then, with the aid of compiler generated mappings, the compiler re-inserted the instrumentation into the post-optimized intermediate representation and generated code. Since this compiler only supported C, it has not yet been incorporated into Tau. As another approach for reducing overhead, Tau experimentally supports dynamic instrumentation via DynInst [10]. Dynamic instrumentation is more promising since it does not inhibit compiler transformations and can monitor object code for which there is no source. Nevertheless, monitoring every call is inherently more expensive than sampling and call path information is limited without static program structure.

Looking forward, we believe that our methods have applicability beyond node-based performance tuning. Since the emergence of petascale computing is being fueled by even larger scales of parallelism than exist on today’s machines, scaling issues will be critical to achieving high performance. Call path profiles can be exploited to detect scalability problems in parallel programs. Coarfa’s ‘scalability analysis’ [14,16] aggregates data from per-process call path profiles of parallel programs to expose scalability bottlenecks in context. Call path profiles enriched with program structure would enable scalability analysis to pinpoint scaling problems to loops and make sense of scaling problems within procedures containing inlining.

Another area of interest is recovering program structure information for *dynamically* compiled code. Languages such as Java that allow objects to change at run time are particularly suited to dynamic compilation because only at run time is enough information known to determine that a class member function is not overridden and is therefore inlinable. Sun Microsystem's HotSpot virtual machine (VM) uses a just-in-time compiler to dynamically compile and 'uncompile' frequently executed portions of the byte code to object code. The effect is similar to loading and unloading dynamically linked libraries, except that the object code has no corresponding image on permanent storage and that object code may be *regenerated* to account for dynamic changes. Because Java's debugging information is file and line based, analysis of the code is required to recover program structure. To support Java profiling tools, Sun developed the Java Virtual Machine (JVM) Tool Interface (formerly known as the JVM Profiling Interface) [33] which enables the construction of either instrumentation or sampling-based call path profilers [18,47]. The JVM Tool Interface provides call backs when a compiled method is loaded or unloaded (Compiled Method Load/Unload), enabling a profiler to copy the object code and its corresponding line map to permanent storage for later analysis. One interesting fact about the HotSpot compiler is that because inlining changes the run time call stack and interferes with Java's ability to maintain security guarantees, the HotSpot VM effectively maintains two call stacks, an optimized stack and a 'shadow' stack corresponding to the source code. This implies that the HotSpot compiler maintains some sort of mapping between the object and byte code to construct the shadow stack. However, the JVM Tool Interface does not appear to make this information available.

Finally, if the recent interest in automatic performance tuning is to be successful, performance information is required that can guide a *program*. If a performance tool cannot even aid an expert human analyst in diagnosing whether there is a performance

problem — and if so, its severity and cause — there is little hope for an auto-tuner. Although our methods do not directly enable auto-tuning, inasmuch as they provide novel information that helps an analyst quickly interpret the performance data, they are a minimal prerequisite.

Bibliography

- [1] V. S. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 50, New York, NY, USA, 1995. ACM Press.
- [2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.
- [3] Apple Computer. Shark. <http://developer.apple.com/performance>. 10 April 2007.
- [4] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.
- [5] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report 21789, IBM, 1999.
- [6] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM Press.

- [7] A. R. Bernat and B. P. Miller. Incremental call-path profiling. *Concurrency and Computation: Practice and Experience*, 2006.
- [8] D. Bonachea. GASNet specification v 1.1. Technical Report UCB/CSD-02-1207, University of California at Berkeley, 2002.
- [9] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 1992. ACM Press.
- [10] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
- [11] F. Cantonnet and T. El-Ghazawi. UPC performance and potential: A NPB experimental study. In *Proceedings of Supercomputing 2002*, Baltimore, MD, November 2002.
- [12] J. H. Chen. Sandia internal report. Technical report, Sandia National Laboratories, 2005.
- [13] W.-Y. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC compiler. San Francisco, California, June 2003.
- [14] C. Coarfa. *Portable High Performance and Scalability for Partitioned Global Address Space Languages*. Ph.D. dissertation, Department of Computer Science, Rice University, January 2007.

- [15] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and Chavarria-Miranda. An evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2005)*, Chicago, Illinois, 2005.
- [16] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. Scalability analysis of spmd codes using expectations. In *To be published in Proceedings of the 21st ACM International Conference on Supercomputing*, Seattle, WA, 2007.
- [17] Compaq Information Technologies Group. *Using Compaq C++ for Tru64 UNIX and Linux Alpha*. Hewlett-Packard Company.
- [18] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM Press, 2004.
- [19] T. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC specifications. <http://upc.gwu.edu/documentation.html>, 2003.
- [20] Free Standards Group. DWARF debugging information format, version 3. <http://dwarf.freestandards.org>. 20 December 2005.
- [21] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.
- [22] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *GCC Summit '06: Proceedings of the GCC Developers' Summit, 2006*, pages 21–36, 2006.

- [23] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 29(5), 1999.
- [24] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.
- [25] R. J. Hall. Call path refinement profiles. In *IEEE Transactions on Software Engineering*, volume no. 6, 1995.
- [26] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the USENIX Summer Technical Conference*, 1993.
- [27] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.
- [28] 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 Trans. Math. Softw.*, 31(3):397–423, 2005.
- [29] Hewlett-Packard. The hiprof profiler. <http://research.compaq.com/wrl/projects/om/hiprof.html>.
- [30] Intel Corporation. Intel VTune performance analyzers. <http://www.intel.com/software/products/vtune>.
- [31] Jefferson Lab. The Chroma library for lattice field theory. <http://usqcd.jlab.org/usqcd-docs/chroma>.

- [32] Lawrence Berkeley Lab. Chombo: A distributed infrastructure for parallel calculations over block-structured, adaptively refined grids. <http://seesar.lbl.gov/ANAG/chombo>.
- [33] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 229–240, 1999.
- [34] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, F. Juelich, R. Rivenburgh, C. Rasmussen, and B. Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.
- [35] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.
- [36] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002.
- [37] G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 194–205, Berlin, Germany, 10–12 1994. Springer-Verlag.
- [38] S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. Ph.D. dissertation, University of Oregon, August 2001. TH/PA/06/22.

- [39] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.
- [40] Silicon Graphics Computer Systems. MIPS extensions to DWARF version 2.0. ftp://ftp.sgi.com/sgi/dev/davea/mips_extensions.ps.
- [41] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 196–205. ACM Press, 1994.
- [42] M. M. Strout, J. Mellor-Crummey, and P. Hovland. Representation-independent program analysis. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 67–74, New York, NY, USA, 2005. ACM Press.
- [43] R. H. Subhash Saini, Johnny Chang and H. Jin. A scalability study of Columbia using the NAS parallel benchmarks. Technical Report NAS-06-011, NASA Advanced Supercomputing Division, 2006.
- [44] UNIX International. DWARF debugging information format. <http://dwarf.freestandards.org>. 27 July, 1993.
- [45] D. A. Varley. Practical experience of the limitations of gprof. *Software: Practice and Experience*, 23(4):461–463, 1993.
- [46] O. Waddell and J. M. Ashley. Visualizing the performance of higher-order programs. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 75–82. ACM Press, 1998.

- [47] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Java Grande*, pages 78–87, 2000.
- [48] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM Press.