

# TECHNICAL REPORT 01-375

Department of Computer Science  
Rice University  
February 2001

## Tools for Application-Oriented Performance Tuning

John Mellor-Crummey

Robert Fowler

David Whalley

{johnmc,rjf}@cs.rice.edu  
Department of Computer Science, MS 132  
Rice University  
6100 Main Street, Houston, TX 77005-1892.

{whalley}@cs.fsu.edu  
Department of Computer Science  
Florida State University  
Tallahassee, FL 32306-4530.

### Abstract

Application performance tuning is a complex process that requires assembling various types of information and correlating it with source code to pinpoint the causes of performance bottlenecks. Existing performance tools don't adequately support this process in one or more dimensions. We discuss some of the critical utility and usability issues for application-level performance analysis tools in the context of two performance tools, *MHSim* and *HPCView*, that we built to support our own work on data layout and optimizing compilers. *MHSim* is a memory hierarchy simulator that produces source-level information not otherwise available about memory hierarchy utilization and the causes of cache conflicts. *HPCView* is a tool that combines data from arbitrary sets of instrumentation sources and correlates it with program source code. Both tools report their results in scope-hierarchy views of the corresponding source code and produce their output as HTML databases that can be analyzed portably and collaboratively using a commodity browser. In addition to daily use within our group, the tools are being used successfully by several code development teams in DoD and DoE laboratories.

## 1 Introduction

The peak performance of microprocessor CPUs has grown at a dramatic rate due to architectural innovations and improvements in semiconductor technology. Unfortunately, it has become increasingly difficult for applications to achieve substantial fractions of peak performance.

Despite the increasing recognition of this problem, the use of performance instrumentation and analysis tools to analyze and tune real applications is not as widespread as one might expect. In our own research on program and data transformations by optimizing compilers we are highly motivated to analyze, explain, and tune many codes, but it was too cumbersome to use existing tool interfaces in this work. We therefore wrote our own tools to address these issues.<sup>1</sup> In this paper we use our experiences with the design and use of these tools to motivate a discussion of how to make performance tools more useful for users.

---

<sup>1</sup>As of this writing, these tools are also being used to improve production applications by several groups at DoD and DoE laboratories.

## 1.1 Impediments to Use

The principal impediment to wider use of performance tools is the amount of time and effort that an application developer needs to expend using these tools to solve performance problems in an analyze/tune cycle. Manual tasks perceived as annoying inconveniences when the tool is applied once become unbearable costs when done repetitively. The main causes of excess user effort are shortcomings in the tool’s explanatory power, *e.g.* they either do not present the information needed to solve a problem, or they fail to present the information in an easily understood form. In either case, the developer/analyst must make up the difference through manual analysis. Aspects of this problem include:

*Performance measures must be related to relevant units in the code, preferably in the original source program.* If performance metrics are reported at the wrong granularity, either the analyst has to interpolate or aggregate information by hand to draw the right conclusions. Data must be presented at a fine enough level, but aggressive compiler optimization combined with instruction level parallelism and out-of-order execution put a lower bound on the size of program units to which a particular unit of cost can be uniquely charged.

*Any one performance measure produces a myopic view.* Large amounts of time spent in a particular program unit are a performance problem only if the time is spent inefficiently. If there is a problem, other measures are necessary to diagnose its causes. Conversely, measures such as cache miss count indicate problems only if the *miss rate* is high and the latency of the misses isn’t hidden.

*Event counts are seldom the measures of interest.* Derived measures such as cache miss ratios, cycles per floating point operation, or differences between actual and predicted costs are far more important.

*Data needs to come from diverse sources.* Hardware performance counters are valuable, but so are “ideal cycles” produced by combining output from a code analysis tool that uses a model of the processor with an execution time profiling tool [5]. Cross-system comparisons are valuable. Either tools need to support data combination and comparison or the analyst has to do it manually.

*Generality and portability are important.* Tools should not be restricted to a narrow set of systems and applications.

The focus of our effort has been to develop tools that are easy to use and that provide useful information rather than inventing new performance measures or new ways to collect measurements.

## 1.2 Our Approach

The observations in the previous section should not be construed to be a set of *a priori* design principles, rather they are the product of our experiences with the tools described here. Our first tool, *MHSim*, is a multi-level memory hierarchy simulator that we built to help analyze the effects of code and data transformations. Analyzing printed output from *MHSim* by hand was too tedious, so we modified the simulator to correlate its results with source code and produce hyper-linked HTML documents that can be explored interactively using Netscape Navigator. While *MHSim* proved to be extremely useful, it had three shortcomings: (1) memory hierarchy event counts alone offer a myopic viewpoint—what is important is whether these misses cause stalls or not, (2) in many cases the simulator was overkill because the similar, though less detailed, information could be obtained at far less expense using hardware performance counters, and (3) many performance problems are not simple matters of memory hierarchy issues. We therefore built a second tool, *HPCView* that correlates source code with data from multiple, diverse instrumentation sources. Like *MHSim*, *HPCView* generates a database of hyper-linked HTML documents that can be explored interactively using Netscape Navigator.

Our use of a standard browser interface has three key advantages. First, it provides users with a familiar interface that gets them started quickly. Second, using Navigator eliminated the need to develop a custom user interface. Third, Navigator provides a rich interface that includes the ability to search, scroll, navigate hyper-links, and update several panes of a browser window in a coordinated fashion. Each of these capabilities facilitates exploration of the web of information about program performance.

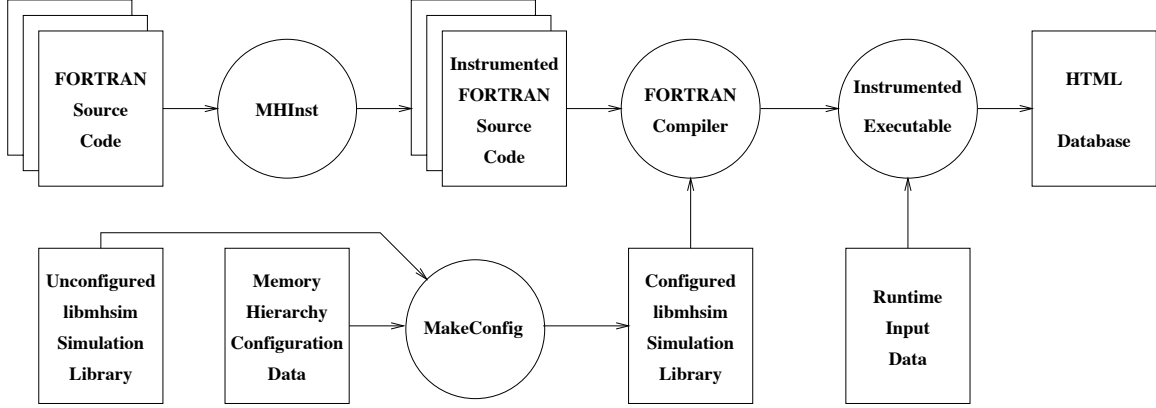


Figure 1: Overview of Using the *MHSim* Simulator

## 2 MHSim

For an application code to achieve high performance, it must exploit caches effectively. Most scientific codes in production use were developed for vector processors that had no caches. When porting such applications to machines with multiple layers of caches, it is difficult to understand the reasons for poor memory hierarchy utilization.

To address this problem we have developed *MHSim*, an integrated simulator and instrumentation tool for Fortran programs. *MHSim* was designed to identify source program references causing poor cache utilization, quantify cache conflicts, temporal reuse and spatial reuse, and correlate simulation results to references and loops in an application program.

Using the *MHSim* simulator to understand the memory hierarchy utilization of a Fortran program involves a sequence of five steps, which are shown in Figure 1. First, a custom configuration of the *libmhsim* simulator is generated based on a specification of the memory hierarchy characteristics of the desired target system. Second, a Fortran source code under study is instrumented using *MHInst*, a source-to-source instrumentation tool. *MHInst* augments the Fortran programs with calls to the *libmhsim* library routines to monitor data accesses and relate simulator results back to the source code. Third, the instrumented Fortran code is then compiled with any Fortran compiler and linked with the *libmhsim* library. Fourth, the compiled version of the instrumented program is executed normally on any platform. During execution, the calls added by the instrumenter will pass information about the program's data accesses to the simulator, which will track information about the program's memory hierarchy utilization separately for each source-program reference. When the program terminates, the simulator correlates the simulation results with the source program and writes them out as a hyper-linked HTML database that forms the basis of a multi-pane user interface. Finally, a user loads the root page of the simulation results and interactively inspects them with Netscape Navigator.

### 2.1 Using the Simulator

A user specifies a number of parameters about the memory hierarchy of the target system being simulated in an *MHSim* configuration file. These parameters include the number of levels of memory hierarchy, the number of cache lines, the line size, the associativity, write-through or write back, and translation-lookaside-buffer configuration information. A simulator configuration tool uses the data in the configuration file to produce header files for a custom configuration of the simulator. A custom instantiation of the *libmhsim* library is produced by compiling the library with the generated header files. This approach was taken to enhance the speed of the simulation. The code to simulate an access to a particular cache is coded as a C++ template. The configuration tool specializes the `cache_access` template for each level of the memory hierarchy and passes the configuration constants for that level of the memory hierarchy as parameters to the template. The C++ template instantiator uses these constants to instantiate a customized version of the cache simulation routine for each level of the memory hierarchy. The resulting code has all of the cache

configuration parameters as compile-time constants, which enables the compiler to optimize the simulator's bit manipulation operations more effectively to improve simulation speed.

The *MHInst* tool instruments Fortran programs to monitor their memory hierarchy utilization by adding calls to the *libmhsim* library to record source-code mapping information for every array reference, loop, and procedure. Loop entry and exits are instrumented with calls to indicate that a scope is being entered or exited, respectively. Each scope entry/exit operation is passed a handle that identifies the source location of the scope. These calls perform bookkeeping for constructing loop level summary statistics. The instrumentation tool adds calls to *MHSim* before each array access to simulate the response of the memory hierarchy. Each simulator call is passed a type code indicating whether the operation is a READ or WRITE access, the number of bytes accessed, the address of the access, and the handle for the source reference corresponding to the access. The simulator uses these handles to relate accesses to source program references and program references to their enclosing scope.

## 2.2 Limitations of Source-Code-Based Instrumentation and Simulation

The *MHSim* simulator uses the addresses passed to the instrumentation routines as the basis for its simulation. The declarations of handles that are inserted by the instrumentation tool perturb the addresses of the program variables. This perturbation affects the absolute position of the data elements which affects the cache conflicts noted by the simulator. Our experience is that the perturbation does not substantially affect the qualitative nature of the results, although in the worst case it could.

A second limitation of the source-code instrumentation strategy used by *MHInst* is that the data accesses specified in the source program will be simulated in their canonical execution order. *MHInst* does not account for any compiler-based transformations that may change the order in which memory references are performed.

A final limitation of source-code-based instrumentation is that many memory accesses are not simulated. These include accesses to scalars that are not allocated to registers and accesses associated with procedure calls, such as saving registers upon procedure entry and restoring registers upon procedure exit. However, Fortran programs, which *MHInst* is designed to instrument, typically have a loop-centric structure and array accesses dominate these other types of accesses.

## 2.3 Exploring *MHSim* Simulation Results

The *MHSim* simulator records the results of its simulation in a collection of HTML and JavaScript files that can be browsed using Netscape Navigator. The top level display of the simulator results is shown in Figure 2.

The top left pane of the display lists the instrumented source files. For this experiment, there was one file: *sweep.f* that contains the computational core of the ASCI Sweep3D neutron transport benchmark. We only instrumented this one file because 98% of the serial execution time is spent in the function *sweep* defined in this file.

The upper right pane displays the source code of *sweep.f*, annotated with hyperlinks for a reference or loop. Clicking on a # hyperlink preceding an array reference will autoscroll each of the panes below to display the simulation results associated with that reference. Next to each scope are two hyperlinks. The 'S' hyperlink will cause loop summary information to be displayed in the panes below rather than the reference-level information shown in the figure. The 'A' hyperlink will display loop-level summary information for each array referenced in the loop.

The next three panes show simulation results collected for the target memory hierarchy, which in this case consists of a TLB, primary and secondary cache. Each level of the memory hierarchy shows the name of the associated source code reference. Clicking on the hyperlink preceding any reference in these panes will cause the source pane to navigate to the appropriate line in the appropriate source file and all other panes of simulation results to auto-scroll to display the information associated with this reference. For each reference, the panes for each memory hierarchy level show the following information:

- The total number of hits associated with this reference.
- The total number of misses associated with this reference.

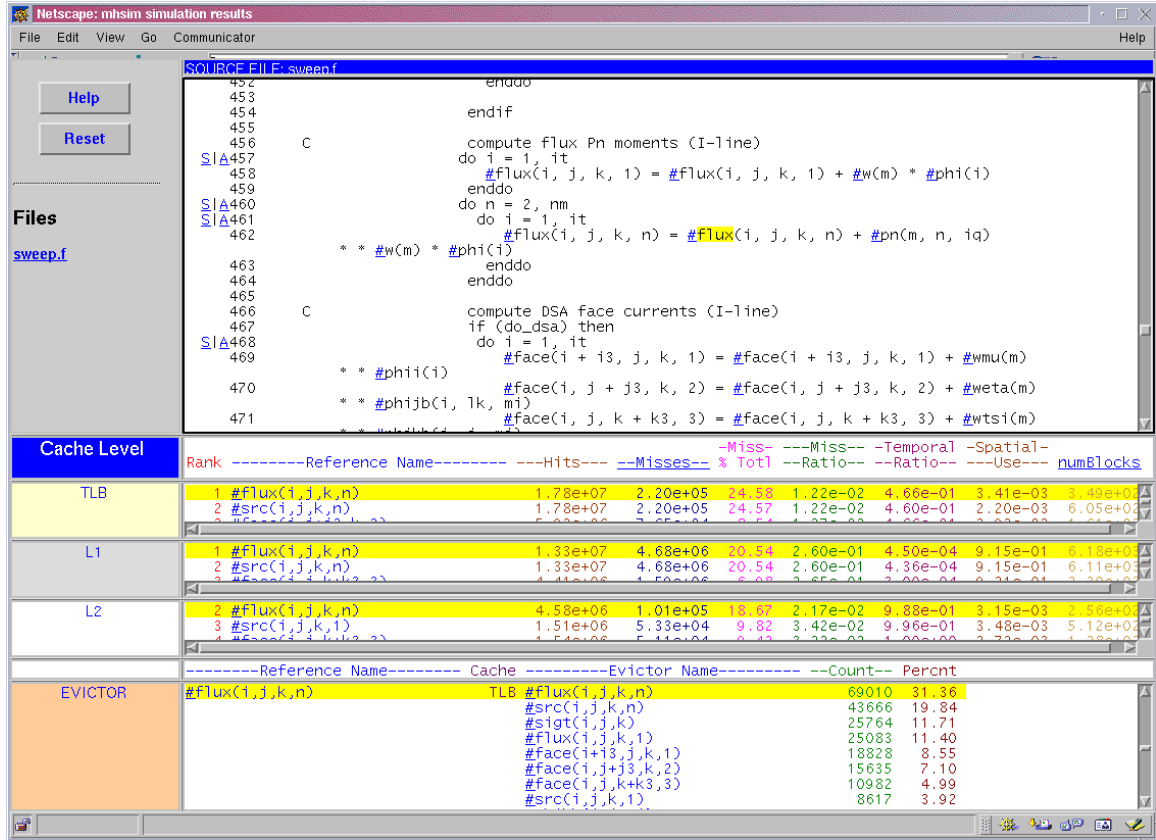


Figure 2: The *MHSim* user interface.

- The number of misses as a percentage of the total misses at this level of the memory hierarchy. The source references are listed in descending order based on this value. Thus, the references causing the greatest percentage of misses are presented to the user first.
- The miss ratio for this reference.
- The fraction of the data reuse for this reference attributed to temporal locality (i.e., number of temporal hits / number of total hits). A hit is due to temporal locality if the bytes referenced in the block were already previously referenced since the block was last brought into that level of the memory hierarchy. It is often useful for the user to know how much of the hit ratio was due to temporal or spatial locality.
- The spatial use (i.e., used bytes / (block size \* number of evictions)). Each time a block is evicted, we store the number of bytes used in the block. The spatial use represents the average fraction of bytes used in the block associated with a reference at the point the block is evicted. If a reference has low spatial use, then this indicates that the machine is wasting cycles bringing in data that is never referenced.
- The number of distinct blocks associated with the reference at this level of the memory hierarchy.

The sample display included shows that the source program reference `flux(i,j,k,n)` on line 462 accounts for over 18% of the misses at each of the levels in the memory hierarchy.

The bottom pane in the top-level window shows evictor information. For a particular source code reference, the evictor pane shows which source-code references caused a cache line containing this reference to be evicted from cache. This information is useful for diagnosing program and data organization problems. With evictor information, a user can quickly determine which source references are competing for the same cache lines. Sometimes a source reference can evict itself, which can occur when the size of the array is

larger than that level of the memory hierarchy. The percentage of the evictions caused by each source code reference is shown. Clicking on the hyperlink for an evicting reference will auto-navigate the source and memory hierarchy panes to show the information available for the evicting reference.

Not shown is the array-level summary information associated with each instrumented scope (loop or procedure) that can be brought up in a separate window by clicking on the 'A' next to the line defining that scope. This summary pane displays the same types of information reported for references, but summarizes all references to each array (and its evictors) within the scope. Examining the summary information presented in the array pane for a costly procedure, or the top-level program scope can identify what array or arrays result in the most misses in the memory hierarchy. These arrays are the appropriate focus for improving a program's overall performance.

### 3 HPCView

The *HPCView* tool was designed to facilitate performance analysis and program tuning by displaying and combining performance measurements from diverse sources and by correlating the results with the program source code. Performance data manipulated by *HPCView* can come from any source, as long as a filter program can convert it to the required input format. To date, the principal sources of input data for *HPCView* have been hardware performance counter profiles. These are generated by setting up the performance counter of interest (*e.g.*, primary cache misses) to generate a trap when it overflows and then histogramming the program counter values at which these traps occur. SGI's *ssrun* and Compaq's *uprofile* utilities both collect profiles this way on MIPS and Alpha platforms, respectively. Currently, we use vendor-supplied versions of *prof* to turn the PC histograms into line-level statistics. We then filter *prof* output into our vendor-independent form using a Perl script. Any information source that generates profile-like output can be used. For example, to analyze excessive register spills in a loop over 3000 lines long, we wrote a script that inspects MIPS assembly code to identify register spill/reload operations and uses source line mapping information to correlate these operations with the program source.

#### 3.1 The HPCView User Interface

A principal design objective of the *HPCView* interface was to present multiple performance metrics in an easily understood, browsable form. The generation of an HCPView dataset is controlled by a configuration file that specifies paths to the source code, a set of files containing performance data, expressions for generating derived metrics, and some parameters that control the appearance of the display. *HPCView* reads specified performance data files, generates the derived metrics, and correlates all of the information to the source code. The result is a hyper-linked database of HTML documents and JavaScript routines that define a multi-pane user interface. Hyper-links cross-reference source code lines with corresponding lines in several performance data tables, and visa versa. This database is then explored interactively using Netscape Navigator. A screenshot of the tool displaying the data for a benchmark program is shown in Figure 3.

The *HPCView* interface consists of a set of panes in a single browser window. On the left side a pane contains links to all of the source files, grouped by directory, for which there is performance information<sup>2</sup>. Clicking on a file name in pane cause the file to be displayed the source file pane.

The source-file pane displays an HTML version of the current source file. The code is augmented with line numbers and with hyper-links to navigate to the performance data tables. Clicking on one of these links navigates the performance data panes to the correct places and highlights the data. If no source code is available, *HPCView* summarizes performance data at the procedure level.

The pair of panes at the bottom right of the window display performance metrics. The upper of the two performance data panes is a flat table displaying line-level data across the entire program. The table is sorted in decreasing order for the performance metric whose header has been selected and highlighted in red. Clicking on the 'sort' link of a metric's column header will re-sort the data for both the performance table and the scope hierarchy display described below. The interface highlights a row if you click on the row's location field. It also navigates the source pane to highlight the selected line.

---

<sup>2</sup>The "Other Files" section lists files for which source code was not found using the search paths of the configuration file.



maximum line yyy.

To perform such aggregation in a compiler and language independent way, we constructed *bloop*, a prototype tool for analyzing application binaries to determine its loop nesting structure using the Executable Editing Library (EEL) [6]. Using the EEL infrastructure, *bloop* builds a control flow graph for each procedure, identifies natural loops, uses interval analysis to interpret their nesting structure, examines the basic blocks within loops to determine the relationship between source statements and the loop nesting structure, and outputs a scope tree representation as an XML file. *HPCView* uses the scope tree file to guide the data aggregation from the statement level to the loop, procedure and file levels.

To cope with the control flow found in a program after it has been radically reorganized by compiler transformations such as software-pipelining or loop fission, *bloop* uses information about program source lines to disentangle the control flow and construct scope trees that can be related back to the original code. Since performance metrics provided to *HPCView* SGI and Compaq's profiling tools aggregate information at the line level rather than the *line instance level* (loop optimizing transformations may cause a line to appear in the context of multiple loops), we currently aggregate together information for all instances of a statement by fusing their enclosing scopes in a scope tree. One possibility for the future is to write a new data collection tool that distinguishes among distinct line instances.

## 4 Computed Performance Metrics: An Example

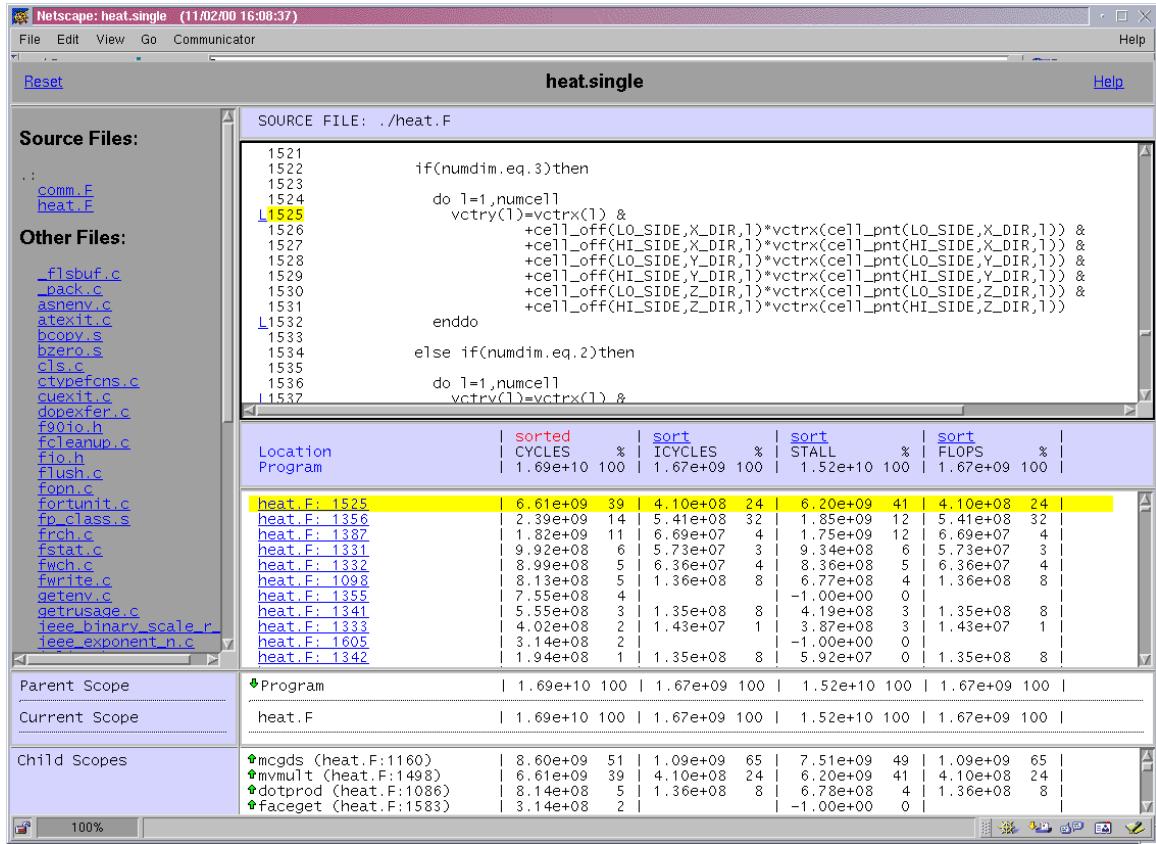


Figure 4: Using HPCView to display both measured and computed metrics.

The metrics shown in Figure 4 were collected on an SGI Origin 2K with R12K processors. The first column in the display shows CYCLES gathered by statistical sampling of the cycle counter. The second column, ICYCLES, shows ideal cycles reported by SGI's pixie utility. The third column, STALL, shows a metric computed by *HPCView* as the difference between CYCLES and ICYCLES. The final column shows



FLOPS (floating point operations) counted by sampling the flop counter. From this display, we see that 41% of the memory hierarchy stall cycles occur in line 1525 of file `heat.F`. The source window shows that this comes from a matrix-vector multiply that uses indirect addressing to index the neighbors of each cell. One potential way to improve performance is to break this loop into nested loops, with the inner loop working on a vector of cells along either the X, Y, or Z axis. This would enable scalar replacement so that successive iterations could reuse elements of vectorx along that dimension.

MTOOL[5] was a tool built specifically to do exactly this kind of analysis. Unlike MTOOL, the mechanism for constructing derived metrics makes *HPCView* easily extensible to do other kinds of analysis.

## 4.1 Approaches to Performance Analysis with HPCView

The ability to bring and analyze data from multiple sources has proven to be useful for a wide variety of tasks. By computing derived performance metrics that highlight differences among performance metrics and then sorting on those differences, we can quickly zero in on phenomena of interest. This capability has been used, as in the example above, to diagnose the cause of performance problems. It has also been used to compare executions of multiple nodes in a parallel computation, to identify input-dependent performance variations, and to perform scalability studies. We have also used it to perform side-by-side comparisons of performance between different architectures and between different implementations of the same architecture.

## 5 Related Work

A number of memory hierarchy simulators have both counted and categorized events. These systems have typically been implemented by computer architects with a bias towards supporting architectural evaluation. *MemSpy* [11] instruments source programs with *Tango* [3], an execution-driven simulator, with calls to the memory simulator for each memory reference associated with heap-allocated (shared) memory or explicitly-identified address ranges in other parts of the code. The granularity of data accumulation is by bins indexed in a 2-D space by code object (procedures) and data-object (data allocated by an instance of a call to `malloc`). Cache misses are classified into compulsory, interference (replaced by data on this processor), and sharing (invalidated by cache coherence operations). The main data display shows the 2-D matrix of bins sorted by code and data units so the most expensive cell appears in the top left corner. Each bin can be examined in more detail. Data at a finer granularity on a per source reference basis is not available.

*CPROF* [9] is a similar simulator, but is based on instrumentation of binary code. In addition, it refines the “interference” miss category into conflict and capacity misses, thus helping to distinguish cases where data re-alignment can help from those where there is just too much data for the cache. The *CPROF* user interface has one pane that displays a source file and another pane which shows lines or data structures sorted in descending order of the number of misses.

The idea of computing the number of cycles lost due to architecture and system overheads has appeared several times in the literature. *MTOOL* [5] estimated the number of cycles that a range of code would take with no cache misses and compared this with the actual execution time. The difference is assumed to be either stalls or time spent in handlers for TLB misses and page faults. To compute the ideal execution time, *MTOOL* instruments executable code by inserting counters to track the number of times each block is executed and it uses a model of the machine to estimate the number of cycles necessary to execute the block. Measurements are aggregated to present results at the level of loops and procedures. While this proved useful for identifying the location of problems, diagnosis was still difficult because the causes of misses and identification of the particular data objects involved was often difficult to determine from *MTOOL*’s output [5].

Going beyond attributing cycles “lost” to the memory hierarchy, *lost cycles analysis* [10] classified all of the sources of overhead (waiting time) that might be encountered by a parallel program. The Carnival tool set [2] extended this into “waiting time analysis”. It provided a visualization tool with each unit of source code having an execution time attributed to it. Colored bars are used indicate the percentage of time spent in each category of overhead.

All recent microprocessors have provided some form of hardware counters that return either cycles, or that count other performance-related events. Profiling using these counters is facilitated by architectures

on which counter overflows can raise exceptions. The most basic way of accessing such profile information is through a text file produced by the Unix `prof` command. Some graphical interfaces are emerging. SGI's *cvperf* [12] performance analysis tool provides a variety of program views. Using *cvperf* one can display only one experiment type, e.g. secondary cache misses, at a time. A pane displaying procedure-level summaries enables one to bring up a scrollable source pane that shows event counts next to each source line. Sandia's *vprof* [7] is another interface that displays a single performance metric with the source code by annotating each line with a count.

*SvPablo* (source view Pablo) is a graphical environment for instrumenting application source code and browsing dynamic performance data from a diverse set of performance instrumentation mechanisms, including hardware performance counters [8]. Rather than using overflow-driven profiling, *SvPablo* library calls are inserted in the program, either by hand, or by a preprocessor that can instrument procedures and loops. The library routines query the hardware performance counters during program execution. After program execution is complete, the library records a summary file of its statistical analysis for each executing process. Like, *HPCView*, the *SvPablo* GUI correlates performance metrics with the program source and provides access to detailed information at the routine and source-line level. Next to each source line in the display is a row of color-coded squares, where each column is associated with a performance metric and each color indicates the importance that source line has on the overall performance of that metric. However, *SvPablo*'s displays do not provide sorted or hierarchical orderings of the program units to facilitate top-down analysis.

## 6 Conclusions

In this paper we described some of the design issues and lessons learned in the construction and use of two performance analysis tools intended specifically to aid in the analysis and tuning of large applications.

*MHSim* was built to help diagnose hard memory hierarchy performance problems and relate them to source code. The evictor and reuse information that we provide is invaluable for such problems. On the other hand, simulation is expensive and since *MHSim* counts events, but does not have a machine specific cost model, it must be used in conjunction with a profiling tool that can help locate and identify the importance of such problems first.

When attempting to tune the performance of a floating-point intensive scientific code, it is less useful to know where the majority of the floating-point operations are than where floating-point performance is low. For instance, knowing where the most cycles are spent doing things other than floating-point computation would be useful for tuning a scientific code. This can be directly computed by taking the difference between the cycle count and the FLOP count for lines, loops or procedures. Our experience analyzing programs with multiple metrics using *HPCView* quickly the need for the tool to compute derived metrics such as cycles per FLOP or miss ratios.

We have found that aggregated information is often much more useful than the information gathered on a per-line and/or per-reference basis. Derived metrics in particular are more useful at the loop level rather than a line level. A key to performance is matching the number and type of issued operations in a loop, known as the loop balance [1], with the hardware capabilities, known as the machine balance. Balance metrics (how many FLOPS per cycle issued versus how many possible, how many bytes per instruction loaded from memory versus peak memory bandwidth per cycle) are especially useful for suggesting how one might tune a loop.

In some cases, we have found that line level (or finer) information can provide misleading information. For example, on a MIPS R10K processor, a counter monitoring L2 cache misses is not incremented until the cycle after the second quadword of data has been moved into the cache from memory. If an instruction using the data occurs immediately after the load, the system will stall until the data is available and the delay is likely to be charged to the second instruction. As long as the two instructions are from the same statement, there's little chance for confusion. However, if the compiler has optimized the code to exploit non-blocking loads by scheduling load instructions from multiple statements in clusters, misses may end up being attributed to the wrong statement. This occurs all too often for inner loops that have been unrolled and software pipelined. The nonsensical fine-grain attribution of costs confuses users. At high levels of optimization, such performance problems are really loop-level issues, and the loop-level information is still sensible. For out-of-order machines with non-blocking caches, per-line and/or per-reference information can

only be useful if some alternative instrumentation technique is used, such as ProfileMe on the Compaq Alpha EV67 processors and successors [4].

We have used the *MHSim* and *HPCView* tools on entire applications. For *HPCView*, this has included a 20,000 line semi-coarsening multigrid code written in C, an 88,000 line Fortran 90 code for three-dimensional fluid flows, and a multi-lingual 200,000 line cosmology application. In each of these codes the tools allowed us to quickly identify significant opportunities for performance improvement. However, for large codes the HTML database size grows large when many metrics are measured or computed. Currently we statically precompute static HTML for the entire set of potential displays for both *HPCView* and *MHSim*. For example, instead of dynamically sorting the performance metric panes, we write a separate copy of the data for each sort order. This includes the data for each of the program scopes. We have seen HTML databases relating several performance metrics to a 150,000 line application occupy 30 megabytes in slightly over 6000 files. To reduce the size of performance databases, we are planning to develop an intelligent browser in Java that can dynamically create views on demand instead of precomputing them all beforehand.

## References

- [1] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.
- [2] Carnival Web Site. <http://www.cs.rochester.edu/u/leblanc/prediction.html>.
- [3] H. Davis, S. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. In *Proceedings of the International Conference on Parallel Processing*, pages 99–107, August 1991.
- [4] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (Micro '97)*, December 1997.
- [5] A. J. Goldberg and J. Hennessy. MTOOL: A Method for Isolating Memory Bottlenecks in Shared Memory Multiprocessor Programs. In *Proceedings of the International Conference on Parallel Processing*, pages 251–257, August 1991.
- [6] E. Schnarr J. Larus. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.
- [7] C. Janssen. The Visual Profiler. <http://aros.ca.sandia.gov/~cljanss/perf/vprof/doc/README.html>.
- [8] D. Reed L. DeRose, Y. Zhang. SvPablo: A Multi-Language Performance Analysis System. In *10th International Conference on Performance Tools*, pages 352–355, September 1998.
- [9] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, October 1994.
- [10] T. LeBlanc M. Crovella. Parallel Performance Prediction Using Lost Cycles. In *Proceedings Supercomputing '94*, pages 600–610, November 1994.
- [11] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *ACM SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.
- [12] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings Supercomputing '96*, November 1996.