Tools for Application-Oriented Performance Tuning

John Mellor-Crummey, Robert Fowler
Dept. of Computer Science
Rice University, MS 132
6100 Main Street
Houston, TX 77005-1892
johnmc,rjf@cs.rice.edu

David Whalley
Dept. of Computer Science
Florida State University
Tallahassee, FL 32306-4530
whalley@cs.fsu.edu

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, other performance measures, such as memory latency, have not kept pace, so 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.[1] 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 application code developers.

To appear in the 15th ACM International Conference on Supercomputing, June 2001, Sorrento, Italy

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 in repeated analyze/measure/tune cycle to solve performance problems. 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. Profiling tools provide information for a small set of program units, typically procedures or source lines (statements). For most performance problems, these are not the right levels of granularity. When programs are compiled at high levels of optimization, the operations of individual statements are interleaved. In this case, the most useful unit for analysis is a loop or some other code block. In other cases, understanding interactions among individual memory references may be necessary. 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. Tools must aggregate data at any level that makes sense for a particular context.

Any one performance measure produces a myopic view. The measurement of one kind of system event seldom identifies or diagnoses a correctable performance problem. Some events

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

measure potential *causes* of performance problems and other events measure the *effects* on execution time. The analyst needs to understand the relationships among these different kinds of measurement. For example, 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 both 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 useful and interesting for performance analysis.

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]. Other code analysis and simulation tools provide forms of information that are not otherwise available. If a code must run on several architectures, it should be easy to use data collected on those systems. Cross-system comparisons are valuable. Either tools need to support data combination and comparison or the analyst has to do it manually.

Tools need to present compelling cases. Performance tools should either identify problems explicitly, or prioritize what look like important problems rather than requiring users to hunt through mountains of printouts or many screens full of data in multiple windows to identify important problems.

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

Manual recompilation and instrumentation are costly. Recompiling an application to insert instrumentation adds to the human cost. Worse, inserting instrumentation manually requires performing consistent modifications to source code. While tolerable for small examples, it is prohibitive for large applications.

Architecture and location independent analysis are important. Vendor supplied tools often work only on the target architecture. This may require that the analysis be done on a machine in the same architecture family as the target machine. This can be problematic when analysts are using a heterogeneous collection of desktop machines and servers.

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.

2. MHSIM

For an application code to achieve high performance, it must exploit caches effectively. Many 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. Even when a program has been specifically designed to use tiling and other locality enhancing techniques to get good cache performance, performance on systems with complex memory hierarchies is often surprisingly disappointing.

To address these problems 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 instru-

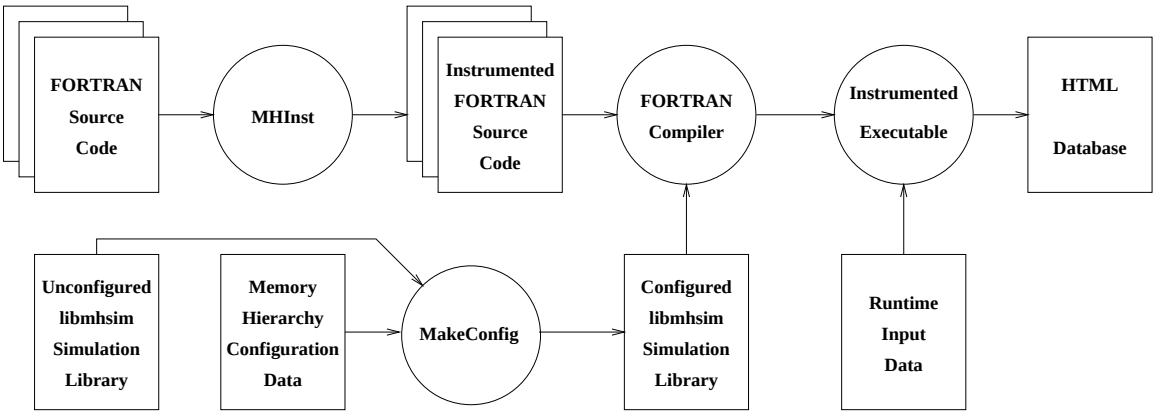

Figure 1: Overview of Using the *MHSim* Simulator

menter 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. Figure 2 shows a Fortran code fragment that has been augmented with calls to the simulator shown in italics. Loop entry and exits are instrumented with calls to indicate that a scope is being entered or exited, respectively. These loop entry and

```
call mhsim_enter_scope(mhsim_scope_15)
do j = 1, jt
   call mhsim_enter_scope(mhsim_scope_16)
   do i = 1, it
   call mhsim(MHSIM_RARRAY, 8, phikbc(i, j, m), &
      mhsim_ref_26, MHSIM_WARRAY, 8, &
      phikb(i, j, mi), mhsim_ref_27, MHSIM_NONE)
   phikb(i, j, mi) = phikbc(i, j, m)
   call mhsim(MHSIM_RARRAY, 8, dj(j), mhsim_ref_28, &
      MHSIM_RARRAY, 8, di(i), mhsim_ref_29, &
      MHSIM_RARRAY, 8, phikb(i, j,mi), &
      mhsim_ref_30, MHSIM_RARRAY, 8, wtsi(m), &
      mhsim_ref_31, MHSIM_NONE)
   leak = leak + &
      wtsi(m) * phikb(i, j, mi) * di(i) * dj(j)
   enddo
   call mhsim_exit_scope(mhsim_scope_16)
enddo
```

Figure 2: A sample loop instrumented using *MHInst*. Original code is in bold face and the instrumentation is in italics.

exits are instrumented with calls to `mhsim_enter_scope` and `mhsim_exit_scope`, 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.

Below we show initialization of a handle for a sample scope.

```
call mhsim_init_scope(mhsim_scope_16, &
    'sweep.mhsim.src.f', 193, '',  &
    MHSIM_LOOP_SCOPE, mhsim_scope_15)
```

This call creates a scope handle known as `mhsim_scope_16`

that corresponds to the loop on line 193 in the generated file *sweep.mhsim.src.f* (a preprocessed version of the source file *sweep.f* generated by the instrumentation tool) which is enclosed in the loop identified by the handle `mhsim_scope_15`. Initialization of a handle for a reference is similar as shown below.

```
call mhsim_init_ref(mhsim_ref_26, MHSIM_RARRAY, &
    'phikbc(i,j,m)', 'sweep:phikbc', &
    mhsim_scope_16, 194, 41, 55, 8)
```

This call creates a reference handle known as `mhsim_ref_26` for a WRITE access to the reference `phikbc(i,j,m)`, an 8-byte element of the local variable `phikbc` declared in routine `sweep`. The reference is on line 194 spanning character positions 41-55. The reference occurs inside the scope `mhsim_scope_16`. 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 3.

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, a primary (L1) cache, and a secondary (L2) 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.
- 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

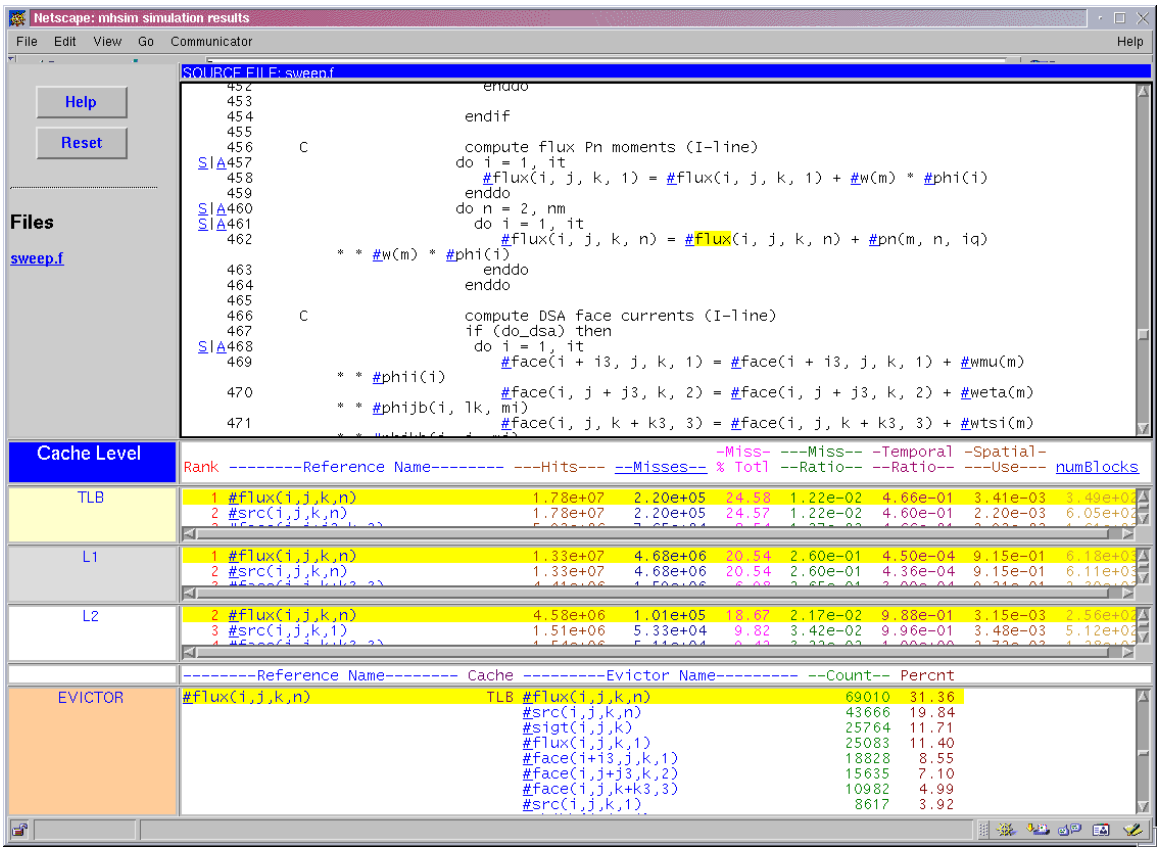

Figure 3: The *MHSim* user interface.

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 autonavigate 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 a standard, profile-like 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 profile-like performance data, expressions for generating derived met-

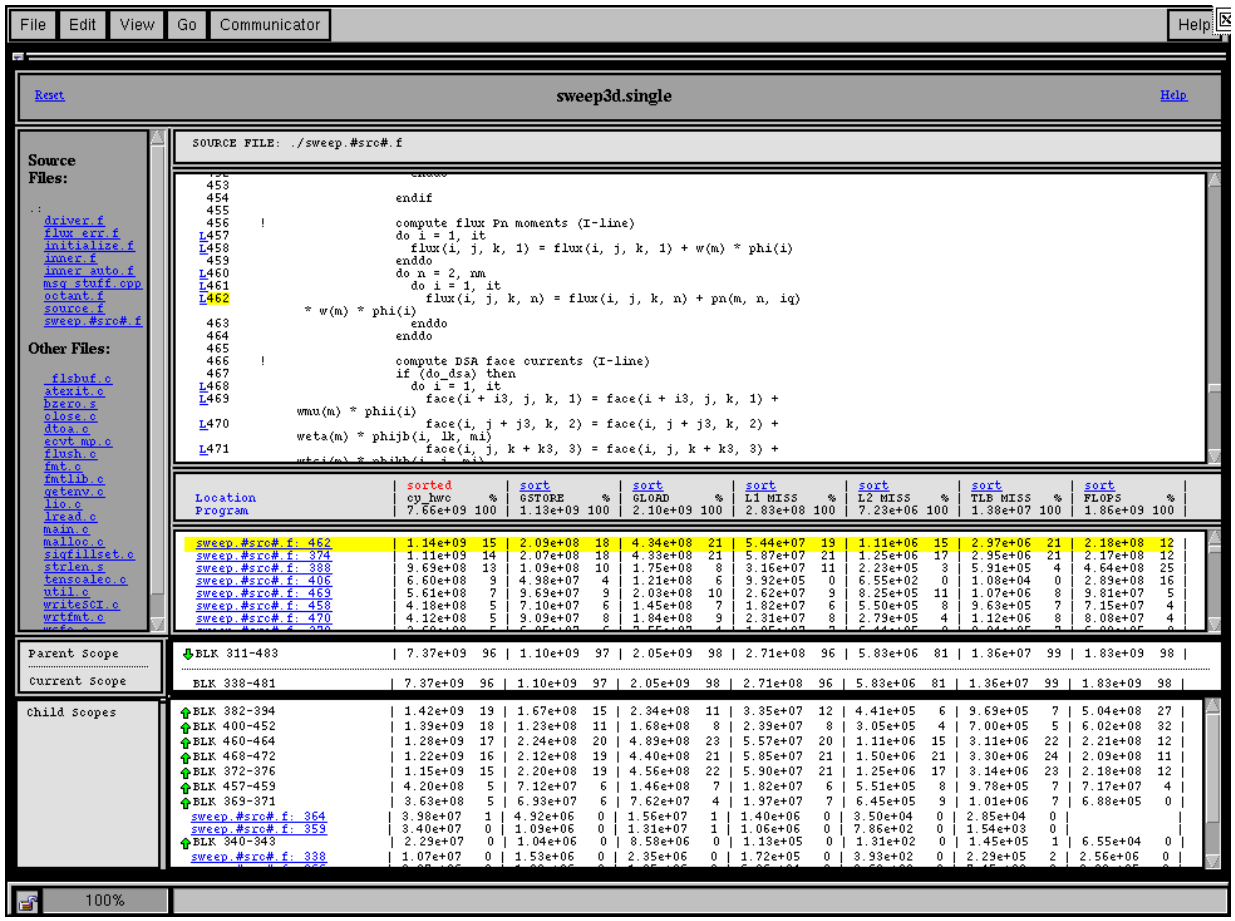

Figure 4: The HPCView user interface.

rics, and some parameters that control the appearance of the display. Optionally, the configuration file can also specify a file that describes the structure of the program as a hierarchical partitioning into syntactically and semantically interesting units. *HPCView* reads specified performance data files, generates the derived metrics, correlates all of the information to the source code at the line level, and aggregates the data into the larger program units. 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[2]. A screenshot of the tool displaying the data for the Sweep3D benchmark program is shown in Figure 4.

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[3]. 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 three panes at the bottom right of the window display performance metrics. The uppermost of the 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. 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.

At the left of each line of the table is a link whose text is the source file name and line number. Clicking on this link scolls the corresponding source file to the correct position and highlights the line. The remaining elements contain

[2]The restriction to use Navigator is due to the use of `layer` constructs to implement highlighting. This will change in future releases.

[3]The "Other Files" section lists files for which source code was not found using the search paths of the configuration file.

values for each of the metrics. Not all lines incur costs for all metrics. For instance, floating point operations and cache misses are often in different lines. In such cases, blanks are left in the table.

Below the flat table is a pair of panes that organize the data hierarchically, with data aggregated by program, source file, procedure, loop[4], and source line. The three-part display shows inclusive performance measurements for the current scope, its parent scope, and its child scopes. Navigation through the hierarchy is done by clicking on the up- and down-arrow icons at the left of each line. The selected scope is moved to the "Current Scope" pane with its parent and children shown above and below it, respectively.

The combination of sorted performance data tables, and easy navigation back and forth between source code and the data tables is the key to the effectiveness of the *HPCView* interface.

3.2 Static Program Analysis

A program's performance is less a function of the properties of a particular source line, rather than the dependences between and balance among the statements in larger program units such as loops. For example, the balance of floating point operations to memory references within one line is not particularly relevant to performance as long as the innermost loop containing that statement has the appropriate balance between the two types of operations and a good instruction schedule. To support loop-level analysis, *HPCView* hierarchically aggregates information from lines and loops to enclosing loops. Thus, the scope hierarchy pane in Figure 4 shows lines marked "`BLK xxx-yyy`" which signifies a loop block with minimum line `xxx` and maximum line `yyy`.

```
<HPCVIEW>
<TITLE name="heat.single" />
<PATH name="." />
<METRIC name="fcy_hwc" displayName="CYCLES">
   <FILE name="heat.single.fcy_hwc.mhf" />
</METRIC>
<METRIC name="ideal" displayName="ICYCLES">
   <FILE name="heat.single.ideal.mhf" />
</METRIC>
<METRIC name="stall" displayName="STALL">
 <COMPUTE> <math><apply><minus/>
    <ci>fcy_hwc</ci> <ci>ideal</ci>
 </apply></math> </COMPUTE>
</METRIC>
<METRIC name="gfp_hwc" displayName="FLOPS">
   <FILE name="heat.single.gfp_hwc.mhf" />
</METRIC>
</HPCVIEW>
```

Figure 5: The *HPCView* configuration file showing the specification of measured and computed metrics shown in Figure 6.

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) [7]. 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* by 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 METRICS

Understanding what opportunities for tuning exist may require computed performance metrics such as the instruction mix in a program's loop nests (loop balance) compared to the ideal instruction mix supported by the target architecture. We have implemented a general mechanism in *HPCView* for supporting computation of derived metrics. Each metric read in from a file has an internal name associated with it. A formula for computing a derived metric in terms of other metrics is a MathML expression [6].

Figure 5 shows the *HPCView* configuration file used to produce the display of measured and computed metrics for the ASCI HEAT benchmark (a 3D diffusion PDE solver) shown in Figure 6. The metrics shown in Figure 6 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 42% 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 con-

[4]Static analysis of the program, either source or executable, is used to identify the loop nesting structure.

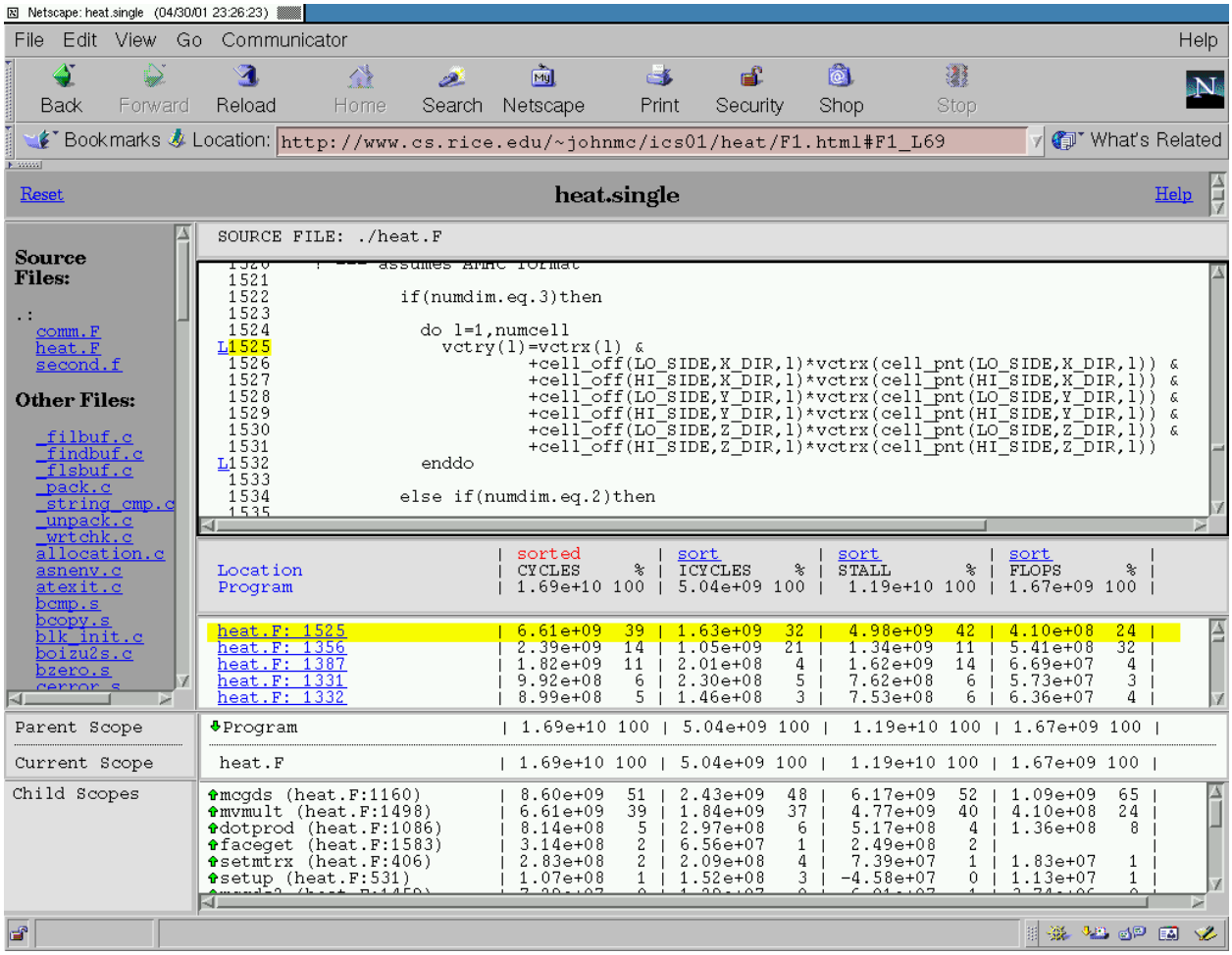

Figure 6: Using HPCView to display both measured and computed metrics.

structing derived metrics makes *HPCView* easily extensible to do other kinds of analysis.

Figures 7 and 8 illustrate the use of computed performance metrics to compare the performance of three different systems: a Compaq ES40 with 500 MHz EV6 processors, a Compaq ES40 with 666 MHz EV67 processors, and an SGI Origin with 300 MHz R12K processors. In this example, we were interested in identifying where in the benchmark program the different systems differed the most in their performance. In Figure 8, the first three metrics defined are the raw cycle count profiles gathered on each of systems. Setting `display=''false''` suppresses the display of these metrics. The next three computed metrics generate cycle counter data normalized to microseconds. Then there are two computed metrics that compare EV6 and EV67 performance by taking their ratio and their difference. Two more computed metrics compare the R12K and EV67. Finally, the `STRUCTURE` construct specifies a file containing the loop structure of the program as extracted using a static analysis of the R12K executable file.

The display in Figure 7 is sorted by the EV6-EV67 column to highlight the widest differences between the systems. In this area of the program, performance is bounded by memory accesses to the flux and phi arrays. Focusing only on costs attributed to line 520, the display indicates that the EV67 is faster than the EV6 by a factor of 1.88 and faster than the R12K by a factor of 1.27. The ratios of performance differences for the enclosing loop, however, are 1.7 and 1.57, respectively. While some part of these differences between line and loop level measurements is due to hardware properties, a substantial contribution to the difference comes from the variation in how the different vendor compilers optimized this loop nest and how they attributed costs to individual source lines. In the absence of accurate attribution of costs (on the R12K, costs may be attributed incorrectly due to out-of-order execution), loop level statistics tend to be more accurate than those at the statement-level. This example illustrates the importance of being able to examine and compare performance measurements aggregated at different granularities.

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

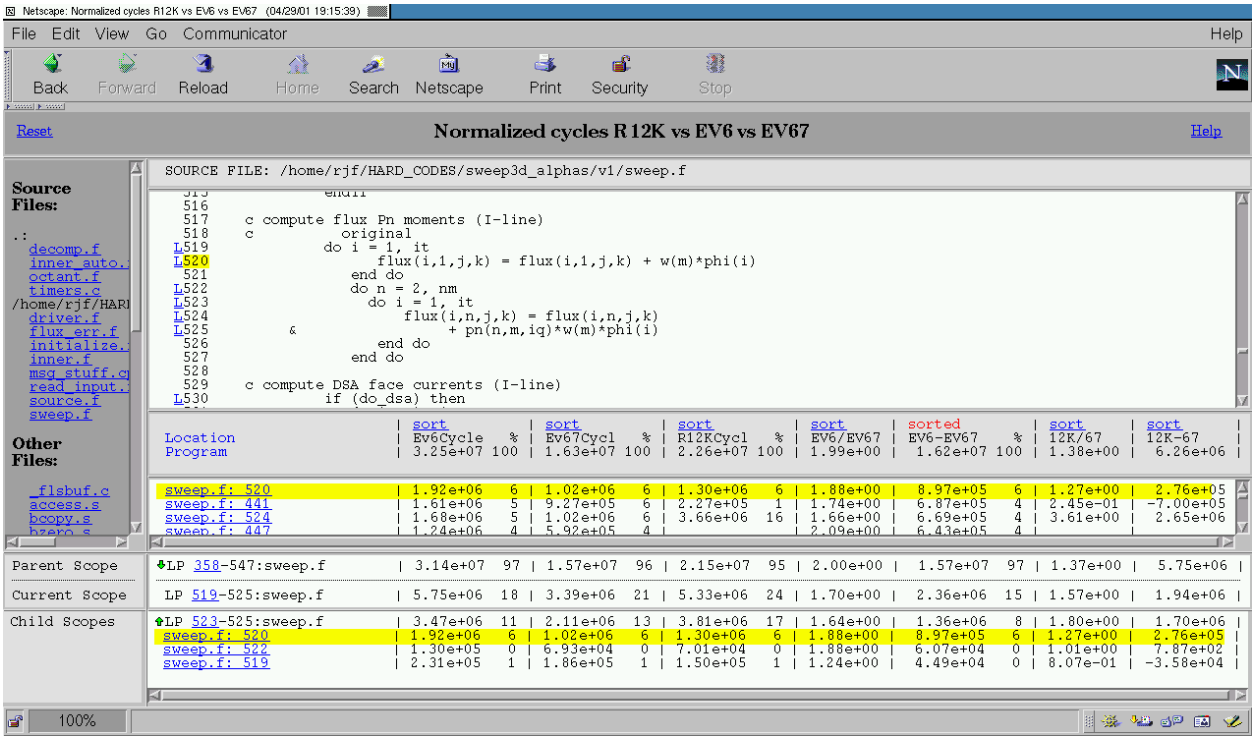

Figure 7: Comparison of cycle counts normalized to microseconds on three architectures.

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* [13] 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 [10] 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.

Performance monitoring can be made more efficient by making use of special hardware features. For instance, the experimental FLASH multiprocessor had a programmable cache controller. *FlashPoint* [12], a performance-monitoring tool, was implemented by extending the cache controller code with simulator-like instrumentation code at a cost of about a 10% slowdown. It gathers data similar to that collected by *MemSpy*, but with the advantage of having direct access to the hardware.

SimOS [14] uses both emulation and simulation techniques. Emulation techniques are based on binary translation to execute applications on a host processor with less than a factor of 10 slower than normal execution time. Simulation techniques can be used when needed to provide more detailed statistics. *SimOS* can dynamically switch between emulation and simulation techniques to allow a user to study portions of long running applications in detail. The data reported is similar to that from *MemSpy* and *FlashPoint*. In particular, none of these tools attempt to measure the reuse of data in caches, nor do they attempt to generate information on the program-level causes of interference such as the identity of evicting instructions.

```
<HPCVIEW>
<TITLE name="Normalized cycles R12K vs EV6 vs EV67 " />

<PATH name="." />

<METRIC name="a" displayName="Ev6Cycles" display="false">
 <FILE name="sweepsing.ev6cy.mhf" /> </METRIC>
<METRIC name="b" displayName="Ev67Cycles" display="false">
 <FILE name="sweep.ev67cy.mhf" /> </METRIC>
<METRIC name="c" displayName="R12KCycles" display="false">
 <FILE name="../sgiv1/mapy.fcy_hwc.mhf" /> </METRIC>

<METRIC name="cy6" displayName="Ev6Cycles" >
  <COMPUTE>  <math><apply>
     <divide/><ci>a</ci><cn>500</cn>
  </apply></math> </COMPUTE> </METRIC>
<METRIC name="cy67" displayName="Ev67Cycles" >
  <COMPUTE> <math><apply>
     <divide/><ci>b</ci><cn>666</cn>
   </apply></math> </COMPUTE> </METRIC>
<METRIC name="cyR12K" displayName="R12KCycles" >
  <COMPUTE> <math><apply>
     <divide/><ci>c</ci><cn>300</cn>
   </apply></math> </COMPUTE> </METRIC>

<METRIC name="r1" displayName="EV6/EV67" percent="false">
 <COMPUTE><math><apply>
    <divide/><ci>cy6</ci><ci>cy67</ci>
 </apply> </math> </COMPUTE> </METRIC>
<METRIC name="d1" displayName="EV6-EV67" >
 <COMPUTE><math><apply>
    <minus/><ci>cy6</ci><ci>cy67</ci>
  </apply> </math> </COMPUTE> </METRIC>

<METRIC name="r2" displayName="12K/67" percent="false">
 <COMPUTE> <math><apply>
    <divide/><ci>cyR12K</ci><ci>cy67</ci>
 </apply> </math> </COMPUTE> </METRIC>
<METRIC name="d2" displayName="12K-67" percent="false">
 <COMPUTE> <math><apply>
    <minus/><ci>cyR12K</ci> <ci>cy67</ci>
  </apply> </math> </COMPUTE> </METRIC>

<STRUCTURE name= "sweep.banal" />
</HPCVIEW>
```

Figure 8: A *HPCView* configuration file used to compute the derived metrics for the display shown in Figure 7.

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* [11] 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 to 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* [15] 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* [8] 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, which include hardware performance counters [9]. 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 illustrates 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.

Acknowledgments

Monika Mevencamp was the principal programmer for the HPCView tool. Xuesong Yuan integrated XML support into HPCView and wrote scripts for translating vendor-specific performance profile formats into the portable XML representation consumed by HPCView. Liwei Peng implemented the HTML interface for MHSIM reports. Nathan Tallent and Gabriel Marin have been the principal implementers of the *bloop* binary analyzer for identifying loops. We thank Jim Larus for letting us use the EEL binary editing toolkit as the basis for constructing *bloop*. This research was supported in part by NSF grants EIA-9806525, CCR-9904943, and EIA-0072043, the DOE ASCI Program under research subcontract B347884, and the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California.

7. 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] W3C Math Working Group. Mathematical markup language (mathml) 1.01 specification, July 1999. http://www.w3.org/TR/REC-MathML.

[7] 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.

[8] C. Janssen. The Visual Profiler. `http://aros.ca.sandia.gov/~cljanss/perf/vprof/doc/README.html`.

[9] 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.

[10] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, October 1994.

[11] T. LeBlanc M. Crovella. Parallel Performance Prediction Using Lost Cycles. In *Proceedings Supercomputing '94*, pages 600–610, November 1994.

[12] D. Ofelt M. Martonosi and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 138–147, May 1996.

[13] 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.

[14] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS machine simulator to study complex systems. *ACM Transactions on Modelling and Computer Simulation*, 7:78–103, January 1997.

[15] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings Supercomputing '96*, November 1996.