HPCView: A Tool for Top-down Analysis of Node Performance

John Mellor-Crummey Robert Fowler Gabriel Marin

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

Abstract

Although it is increasingly difficult for large scientific programs to attain a significant fraction of peak performance on systems based on microprocessors with substantial instruction level parallelism and with deep memory hierarchies, performance analysis and tuning tools are still not used on a day-to-day basis by algorithm and application designers. We present *HPCView*—a toolkit for combining multiple sets of program profile data, correlating the data with source code, and generating a database that can be analyzed portably and collaboratively with commodity Web browsers. We argue that HPCView addresses many of the issues that have limited the usability and the utility of most existing tools. We originally built HPCView to facilitate our own work on data layout and optimizing compilers. Now, in addition to daily use within our group, HPCView is being used by several code development teams in DoD and DoE laboratories as well as at NCSA.

1 Introduction

The peak performance of microprocessor CPUs has been growing 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, and a thirty-year history of papers advocating the use of profiling tools [12] or describing their implementation[13], the everyday use of performance instrumentation and analysis tools to tune real applications is surprisingly rare. Our own research on program and data transformation methods in optimizing compilers has provided us with ample motivation to analyze, explain, and tune many codes, but we found that existing tools were not enhancing our productivity in these investigations. We therefore wrote our own tools to address what we concluded were the impediments to our own work.[1] In this paper we describe HPCView, a performance analysis toolkit that combines generalized profile data captured from diverse sources, correlates that data with source code, and writes a hyperlinked database that can be browsed collaboratively. In addition to describing the key components of the toolkit, we discuss some of the issues that motivated us to begin this project and some of the lessons we have learned.

We believe that the common characteristic of all of the impediments to the effective and widespread use of performance tools is that the tools do not do nearly enough to improve the productivity of the application developers and performance analysts that use them, especially when working on complex problems that require multiple iterations of a measurement, analysis, and code-modification cycle. Specifically, manual tasks perceived as annoying inconveniences when a tool is applied once in a small demonstration become unbearable costs when done repetitively in an analysis and tuning cycle for a big program. In addition to problems that complicate the mechanical process of getting tools to provide useful results, the main causes of excess user effort are shortcomings in the tools' explanatory power. That is, the tools fail to present the information needed to identify and solve a problem, or they fail to present that information in a form that makes it easy to focus on the problem and its solution. In either case, the developer/analyst must make up for the deficiencies of the tool(s) with manual effort.

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

1.1 Key Issues

For performance tools to be widely useful, they must address issues that can be placed in three major categories.

1.1.1 Usability issues related to tool interfaces

Language independence and portability are important. Tools restricted to a narrow set of systems and applications have limited utility. Language- or compiler-based systems have limited applicability for large problems since a large scientific program may have a numerical core written in some modern dialect of Fortran, while using a combination of frameworks and communication libraries written in C or C++.

Architecture and location independent analysis are important. Vendor-supplied tools usuall work only on that vendor's systems. This usually means that the analysis must 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. Furthermore, cross-platform performance studies are extremely useful, but are very difficult to do with vendor tools, each tied to a specific hardware/OS/compiler suite.

Manual intervention for recompilation and instrumentation are costly. Recompiling an application to insert instrumentation is labor intensive. Worse, inserting instrumentation manually requires performing consistent modifications to source code. While tolerable for small examples, it is prohibitive for large applications, especially in a tuning cycle. Inserting instrumentation before compilation must inhibit optimization across instrumentation points or the instrumentation will not measure what was intended. Neither case is acceptable. Inserting instrumentation in aggressively optimized object code is prohibitively difficult.

Tools need to present compelling cases. Performance tools should identify problems explicitly and prioritize what look like important problems, or provide enabling mechanisms that allow the user to search for the problem in a browser. It is unreasonable to require users to hunt through mountains of printouts or many screens full of data in multiple windows to identify important problems. Tools need to assist the user by ranking potential problem areas and presenting a top down view.

1.1.2 Breadth and extensibility of the space of performance metrics

Any one performance measure produces a myopic view. The measurement of time, or of one species of system event, seldom identifies or diagnoses a correctable performance problem. Some metrics 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, while Amdahl's law would have us focus on program units that consume large amounts of time, there is a performance problem only if the time is spent inefficiently. Further, one set of data may be necessary to identify a problem, and another set may be necessary to diagnose its causes. A common example is that measures such as cache miss count indicate problems only if both the *miss rate* is high and the latency of the misses is not hidden.

Data needs to come from diverse sources. Hardware performance counters are valuable, but so are other measures such as "ideal cycles", which can be produced by combining output from a code analysis tool that uses a model of the processor with an execution time profiling tool [6]. Other code analysis and simulation tools generate information that is not otherwise available. If a code must run on several architectures, it should be easy to use data collected on those systems to do cross-system comparisons; either tools will support data combination and comparison, or the analyst will have to do the assembly manually.

Raw event counts are seldom the real 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. While doing the mental arithmetic to estimate such metrics can sometimes be an enjoyable exercise, it eventually becomes annoying. Furthermore, a tool that computes such measures explicitly can then use them as keys for for sorting or searching.

1.1.3 Data attribution and aggregation issues

Data collection tools provide only approximate attribution. In modern multi-issue microprocessors with multiple functional units, out of order execution, and non-blocking caches, the amount of instruction level parallelism is such that it is very difficult or expensive to associate particular events with specific instructions.

On such systems, line level (or finer) information can be misleading. For example, on a MIPS R10K processor, the counter monitoring L2 cache misses is not incremented until the cycle after the second quadword of data has been moved into the cache from the bus. If an instruction using the data occurs immediately after the load, the system will stall until the data is available and the delay will probably be charged to some instruction "related" to the load. 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 be attributed to unrelated instructions from other parts of the loop. This occurs all too often for inner loops that have been unrolled and software pipelined. The nonsensical fine-grain attribution of costs confuses users. On the other hand, 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 accurate or useful if some sophisticated instrumentation such as ProfileMe [5] on the Compaq Alpha EV67 processors and successors is used.

Compiler transformations introduce problems with cost attribution. Many of the things that compilers do to improve performance confound both performance tools and users. Some optimizations, *e.g.* common sub-expression elimination, combine fragments from two or more places in the source program. Other optimizations (*e.g.* tiling) introduce new control constructs. Still other optimizations (*e.g.* loop unrolling and software pipelining) replicate and interleave computations from multiple statements. Finally, "unoptimizing" transformations such as forward substitution of scalar values, or loop re-rolling, may be introduced to reduce register pressure and to facilitate downstream analysis and optimizations.[2] Such transformations, combined with imperfect attribution mechanisms, may move memory references into statements that were carefully designed to touch only values in registers.

With instruction-level parallelism, aggregated properties are vital. Even if profiling instrumentation could provide perfect attribution of costs to executable instructions and if compilers could provide perfect mapping from executable instructions to source code, a program's performance on machines with extensive instruction-level parallelism is less a function of the properties of individual source lines, and more of a function of the dependences and balance among the statements in larger program units such as loops or loop nests[3].

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, a good instruction schedule that keeps the pipelines full, and memory operations can be scheduled to most of the cache miss latency.

We have thus found that aggregated information is often much more useful than the information gathered on a per-line and/or per-reference basis. In particular, derived metrics 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 [2], with the hardware capabilities, known as the machine balance. Balance metrics (FLOPS per cycle issued versus the peak rate, bytes of data loadef from memory per instruction versus peak memory bandwidth per cycle) are especially useful for suggesting how one might tune a loop.

The relevant level of aggregation for reporting performance data is often not known a priori. Typical performance tools report information only for a small, fixed number program constructs, typically procedures and/or source lines (statements). As we argued above, for most performance problems, these are usually *not* the right levels of granularity.

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 available 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 the program unit to which a particular unit of cost can be unambiguously charged. To compensate, tools must flexibly aggregate data at multiple levels.

1.2 Our Approach

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

[2]However, if the downstream analysis/optimization stumbles, the resulting executable may run slower than code compiled with less aggressive optimization.

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. Before starting work on HPCView, we developed MHSim, a multi-level memory hierarchy simulator to help analyze the effects of code and data transformations on cache behavior. Analyzing conventional reports 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. 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 similar, though less detailed, information can be obtained at far less expense using hardware performance counters, and (3) many performance problems are not simple matters of memory hierarchy issues. We began work on *HPCView* to broaden the applicability of the ideas behind the MHSim interface.

A principal design objective of *HPCView* was to correlate source code with data from multiple, diverse instrumentation sources and present the information in an easily understood, interactively-browsable form that facilitates top-down performance analysis. Like *MHSim*, *HPCView* produces a hyper-linked HTML document for interactive exploration with commodity browsers. Our use of a commodity browser interface has three key advantages. First, it provides users with a familiar interface that gets them started quickly. Second, it eliminates the need to develop a custom user interface. Third, commodity browsers provide 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 in the program performance dataset.

2 HPCView

The *HPCView* performance analysis toolkit is a collection of scripts and programs designed to facilitate performance analysis and program tuning by correlating performance measurements from diverse sources with program source code. A program called `hpcview` is at the toolkit's center.

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. Such profiles are generated by setting up a counter of to count events of interest (*e.g.*, primary cache misses), to generate a trap when the counter overflows, and then to histogram the program counter values at which these traps occur. SGI's `ssrun` and Compaq's `uprofile` utilities collect profiles this way on MIPS and Alpha platforms, respectively. Any information source that generates profile-like output can be used as an `hpcview` data source. Currently, we use vendor-supplied versions of `prof` to turn PC histograms collected by `ssrun` or `uprofile` into line-level statistics. Scripts in the *HPCView* toolkit transform these profiles from their platform-specific formats into our architecture-independent XML-based profile format.

To direct the correlation of performance data with program source test, `hpcview` is driven by a configuration file that contains paths to directories that contain source code, specifications for a set of performance metrics, and parameters that control the appearance of the display. The performance metric specifications either point to a file containing measured profile data, or describe a computed metric using an expression in terms of other measured or computed metrics. Optionally, the configuration file can also name a file that describes a hierarchical decomposition of the program, either the loop nesting structure, or some other partitioning.

`Hpcview` reads the specified performance data files and attributes the metrics to the nodes of a tree representing the hierarchical program structure known to the tool, typically files, procedures, nests of loops, and lines. Next, it uses these measures as operands for computing the derived metrics at each node of the tree. Finally, `hpcview` associates the performance metric data with individual lines in the source text and produces a hierarchical hyper-linked database of HTML files and JavaScript routines that define a multi-pane hypertext document. In this document, hyper-links cross-reference source code lines and their associated performance data. This HTML database can be interactively explored using Netscape Navigator 4.x or Internet Explorer 4+.

To help new users on SGI and Compaq machines get started, a generic script called `hpcquick` can orchestrate the whole process from a single command line. As a side effect, `hpcquick` leaves behind the an

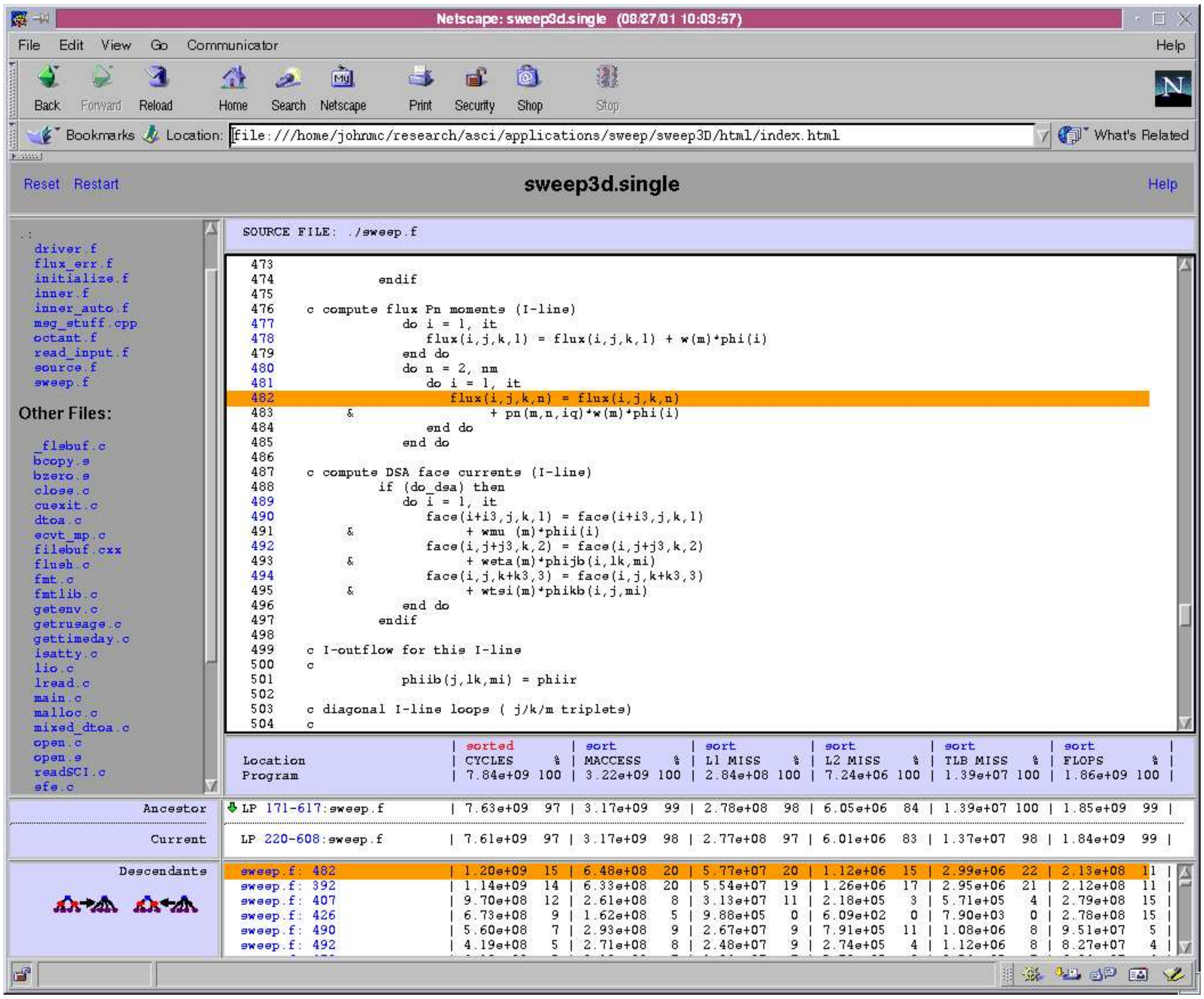

Figure 1: The HPCView user interface.

executable script and a configuration file for the user to modify and extend to perform additional analyses. Later, for production use, we use other scripts that automate the entire process from doing the data-gathering runs to generating the database.

2.1 The HPCView User Interface

The *HPCView* interface consists of a single browser window containing multiple panes. A screenshot of the interface displaying some data for the ASCI Sweep3D benchmark program is shown in Figure 1. The top-left pane contains links to all of the source files, grouped by directory, for which there is performance information. The "Other Files" section lists source files not found using the search paths in the configuration file. Clicking on a file name in the left pane causes the file to be displayed in the source file pane and the performance data panes to display the available metrics for that file as well.

The source-file pane on the top-right displays an HTML version of the current source file. The code is augmented with line numbers and hyper-links used to navigate the two performance data panes at the bottom to display the performance data associates with that line. Hyperlinks in the performance data panes navigate the source pane to the associated source file and line. For functions or files not found using the configuration paths, `hpcview` generates a synopsis in place of the missing code.

The two panes at the bottom present the performance data as a hierarchical partitioning, i.e. a tree.

At the top of the hierarchy is the program. Using a typical structural decomposition, the nodes below the program are files and then procedures. Below a procedure in the hierarchy may be one or more levels of loops. The leaves of the tree are individual source lines for which at least one non-zero performance measure data has been found. Thus, the leaves of the tree are sparse. Not all lines will have values for all metrics. For instance, floating point operations and cache misses often are associated with different lines. Missing data is indicated with a blank.

The tables of data presented in the performance panes are sorted in decreasing order according to the currently selected performance metric. Initially, the leftmost metric in the display is the one selected. Clicking on the 'sort' link of a different metric's column header will display the data sorted by that metric.

When a browser first reads an *HPCView* database, the current scope is the entire program and the source files are shown as its children. 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" pane with its ancestor and descendants shown above and below it, respectively.

To the left of the hierarchical display of the scope tree are iconic links that are used to to adjust the view of the scope hierarchy by "flattening" or "unflattening" the descendants of the current scope. Flattening the current scope replaces each non-leaf scope shown in the descendants pane with its children. Leaf scopes cannot be flattened further, so they are retained as is. Progressive unflattening reverses the process until the descendants pane shows only children of the current scope. Interactive, incremental flattening was added to the current version (2.0) of *HPCView*, and a completely flat line-level pane was removed, specifically to support and encourage top-down performance analysis. When tuning the performance of a scientific program, the most important question often to identify the most costly loop in the program. At the top level, files are initially displayed as the children of the program. After one round of flattening, the program's descendants become the set of procedures. However, knowing the cost of each procedure still doesn't tell us which contains the most costly loop. By flattening one more level, all top-level loops are viewed as peers with the most costly one at the top of the display. Without the ability to flatten a profile in this way, one would be forced to navigate into each procedure to look for the one containing the highest cost loop.

The combination of panes of sorted performance data, the hierarchical structure of the data in the tree, the ability to look at all of the peers at each level of the tree, and simple navigation back and forth between source code and the performance data are the keys to the effectiveness of the *HPCView* interface.

2.2 Computed Metrics

Identifying performance problems and opportunities for tuning may require computed performance metrics that, for instance, compare the instruction mix in a program's loop nests (loop balance) to the ideal instruction mix supported by the target architecture.

When attempting to tune the performance of a floating-point intensive scientific code, it is often 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 is useful for tuning scientific codes. This can be directly computed by taking the difference between the cycle count and FLOP count divided by a target FLOPs per cycle number, and displaying this measure at loop and procedure level. Our experiences with using multiple computed metrics such as miss ratios, instruction balance, and "lost cycles" using *HPCView* underscore the power of this approach.

To support the computation of derived metrics, the configuration file assigns an internal name to each metric as part of its definition. The definition of each computed metric is a MathML [7] expression that refers to previously-defined metrics, either read from a file or computed.

2.3 Program Structure

A program's performance is less a function of the properties of individual source lines than of the dependencies and balance among the statements in larger program units such as loops or loop nests. For example, the balance of floating point operations to memory references within any 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.

As *HPCView* processes its performance profile inputs, it incrementally builds a scope tree that reflects the structure is either read in explicitly or that can be inferred from information reported. The root of the tree represents the entire program. Below the root are nodes representing the files containing source code. Below the files are procedures and, using typical profile data without any additional informantion, below the procedures are nodes representing lines. Some compilers (e.g. SGI) use lines as surrogates for entire statements, while other compilers (e.g. Compaq) provide attribution of costs to individual lines (and columns within lines) within multi-line statements.

To better organize the information available from the performance data, `hpcview` (the program) can also be instructed to initialize the scope tree with information from one or more XML "structure" files (See Figure 2.) that contain a tree representation of a hierarchical decomposition of the program as defined by some external source. The program structure description that we use is extremely general. Here is a list of the possible scopes and the relationships among them.

- *Program* scope – is the root of the tree and can include files and groups.
- *File* scope – can contain procedures, statements, loops, and groups.
- *Procedure* scope – can contain loops, statements, and groups.
- *Loop* scope – may include zero or more loops, statements, and groups.
- *Statement* scope – specifies a contiguous range of source lines.
- *Group* scope – is a catch-all scope that can include zero or more files, procedures, loops, statements, or other groups.

The *group* scope is not bound to any conventional syntactic program unit, but it can include any set of scopes and it can be included in any other scope. The motivation for this virtual unit is to support arbitrary, non-syntactic partitionings. The simplest use is to aggregate non-adjacent statement ranges into a single unit. It can also be used to represent a semantic partitioning of the program in which program units with related function are aggregated for reporting purposes.

Structure files can be generated in several ways:

- They can be emitted by a compiler to reflect the program structure discovered in the source file.
- Structure can be derived from static or dynamic analysis of the executable object code.
- They can be defined explicitly. For example, annotations applied to the code of a library could be to induce a hierarchically partitioning by semantic category.

We initially used a compiler-based structure generator [10] to emit loop informantion. This was limited to Fortran77 programs and required explicitly applying the tool to each source file. We are currently using static analysis of binary executables to extract a control flow graph and to approximate syntactic loop nesting. This is described in the next section. Explicit construction of structure files from program annotations is someting we intend to support in the near future to provide the flexibility to define abstractions that aggregate program units by functional characteristics, or other semantic criteria.

3 Extracting Structure from Executables

The principal disadvantage of analyzing source files to determine a program's structure is that it requires separate handling for each source language. Other issues include the need to analyze many source files in many directories, unavailable source files, and the discrepancies between the structure of source code and the structure of the highly-optimized object code. To address all of these problems, we developed `bloop`, a tool that leverages the Executable Editing Library (EEL) [9] infrastructure to construct a hierarchical representation of an application's loop nesting structure based on an analysis of the executable code.

Figure 2 shows an example of an XML document constructed by `bloop` containing an `hpcview` scope tree encoding an application's structure. The XML document includes scope trees for the procedures

```
<PGM n="/apps/smg98/test/smg98">
   ...
   <F n="/apps/smg98/struct_linear_solvers/smg_relax.c">
      <P n="hypre_SMGRelaxFreeARem">
         <L b="146" e="146">
            <S b="146" e="146"/>
         </L>
      </P>
      <P n="hypre_SMGRelax">
         <L b="297" e="328">
            <S b="297" e="297"/>
            <L b="301" e="328">
               <S b="301" e="301"/>
               <L b="318" e="325">
                  <S b="318" e="325"/>
               </L>
               <S b="328" e="328"/>
            </L>
            <S b="302" e="312"/>
         </L>
      </P>
      ...
   </F>
   ...
</PGM>
```

Figure 2: A partially-elided scope tree produced by `bloop` that shows a nesting of file, routine, loop and statement scopes.

`hypre_SMGRelaxFreeARem` and `hypre_SMGRelax` in the context of an elided scope tree for the semi-coarsening multigrid application to which they belong.

To construct the scope tree representation, `bloop` first builds a control flow graph of the executable by analyzing the control flow machine instructions in each procedure. Next, it uses interval analysis [18] to identify the natural loops in each procedure's control flow graph and to determine their nesting. Then, `bloop` uses object-to-source mapping information recorded by the compiler(s) in the executable's symbol table to map each instruction in each basic block to the corresponding source code location. The combination interval nesting information, the position of each basic block in the interval nesting structure, and source statement locations contained in each basic block enables `bloop` to construct a mapping between source statements and the parts of loop nesting structure in the executable.

The major difficulty with this strategy is that the code transformations used by optimizing compilers result in complicated mappings between object instructions and source statements. Instructions related to a single source program statement often appear in many different basic blocks in the loop nesting structure. For instance, loop-invariant code motion can hoist some fragments of a statement out one or more loop levels. Also, loop optimizations such as software pipelining may split a source loop into multiple object loops—e.g., prolog, steady state, and epilog phases—with each of these object code loops containing instructions mapped to the same range of source lines.

Since each each statement originally appeared uniquely in the program's source code, it is natural to try to represent and view performance data in a form where each source statement appears in only in its original place. Thus, common profiling tools (e.g., SGI and Compaq's `prof` commands) report performance metrics at the source line level without distinguishing among multiple different fragments or instances of the same source line. Similarly, our current source code scope tree representation of a program's structure puts each source construct in at most one place. On the other hand, a single source construct can be mapped to object instructions in several different places in the object-level control flow graph. We therefore developed a normalization mechanism[3] to generate an approximation to the structure in which each source statement

[3] A command line switch to bloop suppresses normalization. This is useful for understanding compiler transformations and

is reported in at most one position. The fragment shown in Figure 2 is an example of this normalized representation.

To normalize the scope tree representation, `bloop` repeatedly applies the following rules until a fixed point is reached.

- Whenever instances of one statement appear in two or more disjoint loop nests, the normalizer recursively fuses their ancestors from the level of the outermost of the statement instances out to the level of their lowest common ancestor. *Rationale: loop transformations often split a loop nest into several pieces; to restore the representation to a source-like nesting requires fusing these back together.*

- When a statement appears at multiple distinct loop levels within the same loop nest, all instances of the statement other than the innermost are elided from the mapping. *Rationale: the statement probably incurs the greatest cost in the innermost loop; moving the costs incurred in outer loops inward will produce an acceptable level of distortion.*

Finally, since the structure produced by `bloop` is intended only to organize the presentation of performance metrics, all of the loops in each perfect loop nest are collapsed into the outermost loop of the nest. This makes it less tedious to explore the scope hierarchy since it avoids presenting exactly the same information for each of the loops in the perfect nest.

One unfortunate consequence of our current reliance on vendor tools for gathering performance profiles is that the reporting of performance metrics at the source line level by these tools can make it impossible in some case to correctly attribute performance metrics. In particular, if a program contains multiple loop nests on the same line (Such a case arises when a macro expands into multiple loop nests.), one cannot distinguish the performance of the individual loop nests. The normalized representation produced by `bloop` for such cases dutifully fuses such loop nests so these aggregate statistics can be reported sensibly. To avoid the inherent inaccuracy when SGI and Compaq's `prof` commands aggregate information to source lines rather than reporting separate information for line instances or fragments, our future plans include writing new data collection tools that distinguish among distinct line instances.

A substantial benefit of using a binary analyzer such as `bloop` to discover program structure is that the results of compiler transformations such as loop fusion and distribution are exposed to the programmer. For example, when loops are fused in the executable, the normalized representation reported by `bloop` will show statements in the fused loop as belonging to a common loop nest.

It is worth noting that any inaccuracy by a compiler in mapping object instructions to source constructs degrades the accuracy with which `bloop` can generate a reasonable normalized scope tree. While mapping information seems to be accurate for the SUN and Alpha compilers, the SGI C compiler sometimes associates incorrect source line numbers with instructions performing loop condition tests and adjusting the values of loop induction variable. We have not yet found a reliable method to compensate for this bug. It affects the reported line numbers of the loop scopes and it sometimes distorts the structure of the normalized tree.

3.1 Research Issues

In its current incarnation, HPCView does not provide any facilities for aggregating performance metrics up the dynamic call chain. We are exploring methods for capturing dynamic call chain information through binary instrumentation. Such experiments are using binary rewriting features of EEL [9].

Another extension for the future is to collect and report performance information for line instances. Combined with the un-normalized control structure information recovered by `bloop`, this would enable more detailed analysis of the effectiveness of aggressive loop transformations. As developers of compiler optimization algorithms, we expect this will be useful to ourselves. We alst believe that this will be useful for application tuning, especially for understanding the relationships among performance, source-level program structure, and and the detailed structure of the code generated at each level of optimization.

---

will be used in a future version of HPCView to display transformed code.

4 Using the Tools

In this section, we briefly illustrate how *HPCView* is used by walking through an analysis and tuning of Sweep3D [8], a 3D Cartesian geometry neutron transport code benchmark from the Department of Energy's Accelerated Strategic Computing Initiative (ASCI). As a benchmark, this code has been carefully tuned already, so the opportunities for improving performance are slim.

The hardware platform used throughout these examples is a single 300MHz MIPS R12000 processor of a 16-processor SGI Origin 2000. Each processor has a 32KB 2-way set-associative primary data cache, an 8MB 2-way set associative unified secondary cache, and a 64-entry TLB that maps a pair of 16KB pages per entry.

To automate the collection of data and the generation of the HPCView display shown in Figure 1, we used a script that runs the program seven times. When a run is initiated, the execution environment is directed to take a statistical sample of some program event during the execution. A separate monitored run was needed to collect the data for each of the columns labeled CYCLES, L1 MISS, L2 MISS, TLB MISS, and FLOPS. The column labeled MACCESS presents the memory accesses, a synthetic metric that was computed as the sum of graduated loads and graduated stores; loads and stores were each collected in a separate monitored run. Once the data was collected, a script ran prof, conversion filters, and, finally, `hpcview` to generate the HTML that correlates the source program each of the metrics.

Each of the numbers in the performance metrics is computed by multiplying the number of samples by the inverse of the sampling rate expressed either as events per sample or events per cycle. A program-wide total for each metric is displayed in the corresponding column header. An initial assessment of the program's performance and the potential for improvement can be performed by comparing these totals. Comparing the total number of cycles to the total number of FLOPs, we see that the program completes a floating point operation once every 4.21 cycles. While this indicates that there is a substantial gap between achieved performance and peak floating point performance, a comparison of between the number of FLOPs and the number of memory accesses shows that there are 1.73 memory operations per FLOP. Since only one memory operation can be performed in each cycle, the number of memory operations places an upper bound on achievable floating point performance unless the program can be restructured to increase temporal reuse of values in registers and reduce the number of memory operations. The ratio between the number of cycles and the total number of memory operations is 2.43. Thus, the program performance could potentially be improved if memory hierarchy utilization can be improved. While tools such as SGI's perfex [19] provide such whole-program metrics and can be used for this sort of analysis, HPCView provides the same perspective for each scope level within a program including files, procedures, and loops.

We continue our overall performance assessment using the totals shown in Figure 1. The primary cache miss rate computed as the ratio of the L1 MISS and MACCESS columns is 8.8%. The secondary cache miss rate (computed as the ratio of L2 misses to L1 misses) is 2.5%. The TLB miss rate is about 0.4%. While there is some room for improvement in memory hierarchy utilization, the miss rates are not particularly excessive at any of the levels in the hierarchy, so the impact of improvements should be modest.

To determine if any source lines were responsible for a large share of the execution time, we navigated to the computational loop responsible for 97% of the execution time and flattened it to display all of the statements inside it as descendants. The top two source lines in the descendants display account for 29% of the program's total execution time. In the column displaying TLB information, we see that the top two lines account respectively for 22% and 21% of the TLB misses in the entire execution. A click on the hyper-link associated with the top line in the performance display caused the source pane to display and highlight the selected line. A quick examination of the most costly source line and its surrounding loops yields the reason for its poor TLB performance: accesses to `flux(i,j,k,n)` are being performed in a loop nest in which the innermost loop iterates over `i` (the stride-1 index), but the immediately enclosing loop iterates over `n`. This iteration order causes the program to access a dense vector, then hop to another page to access another vector. The second line in the performance display point refers to another line in the program accessing `src(i,j,k,n)` in the same fashion. Transforming the program or the data so that the loop nests iterate over the array dimensions left to right will yield the best performance for Fortran's column-major layout.

An examination of the program showed that other than a few lines in an initialization procedure, there were no other accesses to the `src` and `flux` arrays in the code. In this case, permuting the dimensions of these arrays so that they are referenced with `n` in the second dimension (i.e., `src(i,n,j,k)`) will better

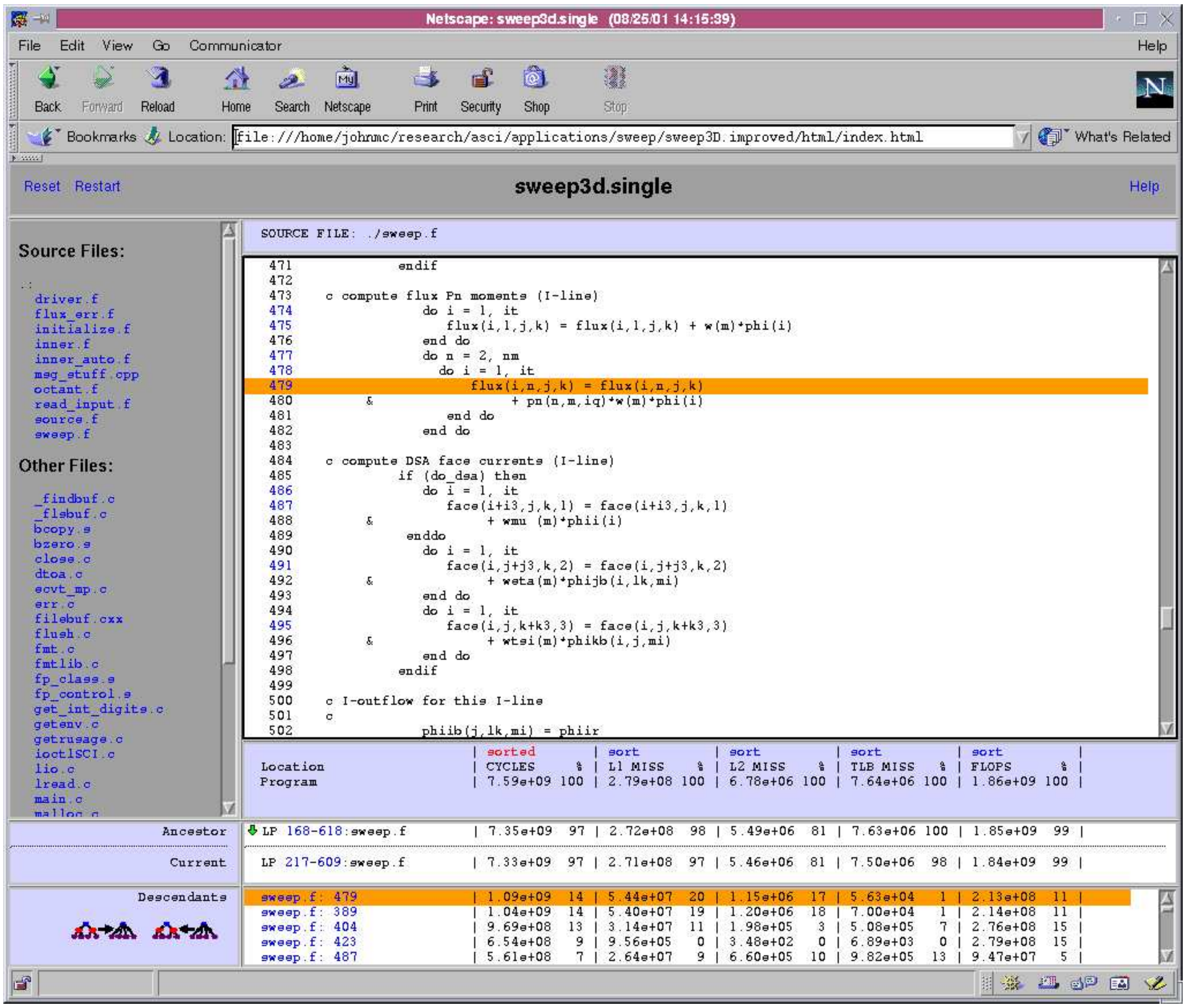

Figure 3: A HPCView display for Sweep3D after permuting dimensions of `src` and `flux`.

match their layout to the access order and avoid excessive page faults. Figure 3 shows the same information for this improved version of the program. The two corresponding lines in the modified program remain the most costly as shown in the line-level performance display (line numbers have changed slightly between versions: 482 is now 479 and 392 is now 389). Comparing the aggregate TLB misses for corresponding pair of lines in these two programs, we see that they have dropped from 43% of misses to only 2%. Despite the improvement in TLB performance, these two lines remain the most costly in the program, accounting for 28% of the execution time, which as a whole is slightly less than 2% faster than the original version.

4.1 Other Examples

HPCView has also been used on several large production codes including CHAD, an 88,000 line Fortran 90 code for three-dimensional fluid flows with chemical reactions and fuel sprays developed at Los Alamos National Laboratory. With CHAD, HPCView helped uncover poor spatial and temporal reuse in a key procedure. Poor reuse occurred because of the vendor Fortran 90 compiler didn't fuse aggressively enough after scalarizing a large collection of Fortran 90 vector statements. Grouping compatible vector statements by hand allowed the compiler to fuse more loops, leading to a 40% reduction in loads.

HPCView has also been used to analyze SMG98, a 20,000 line semi-coarsening multigrid code written in

C that was developed at Lawrence Livermore National Laboratory. Using HPCView it was straightforward to determine that the biggest problem with the code was several of its key computational loops were memory bound. Node performance tuning efforts focused on this problem have led to improvements of 20%.

```
<HPCVIEW>
<TITLE name="heat.single" />
<PATH name="." />
<METRIC name="fcy_hwc" displayName="CYCLES">
   <FILE name="heat.single.fcy_hwc.pxml" />
</METRIC>
<METRIC name="ideal" displayName="ICYCLES">
   <FILE name="heat.single.ideal.pxml" />
</METRIC>
<METRIC name="stall" displayName="STALL">
 <COMPUTE>
  <math>
    <apply><max/>
      <apply><minus/>
        <ci>fcy_hwc</ci> <ci>ideal</ci>
      </apply>
      <cn>0</cn>
    </apply>
  </math>
 </COMPUTE>
</METRIC>
<METRIC name="gfp_hwc" displayName="FLOPS">
   <FILE name="heat.single.gfp_hwc.pxml" />
</METRIC>
</HPCVIEW>
```

Figure 4: The *HPCView* configuration file showing the specification of metrics displayed in Figure 5.

Figure 4 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 5. The first column in the display shows CYCLES gathered by statistical sampling of the cycle counter. The second column, ICYCLES, is "ideal cycles" as computed by SGI's pixie utility. The third column, STALL, shows a metric computed by *HPCView* as the maximum of zero and the difference between CYCLES and ICYCLES. MTOOL[6] was a tool built specifically to do exactly this kind of analysis. In contrast to MTOOL, however, HPCView's mechanism for constructing derived metrics makes it easy to add other kinds of analyses to the display. The final column of the figure shows graduated floating point instructions. 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 *vctrx* along that dimension.

Figures 6 and 7 illustrate the use of computed performance metrics to compare the behavior 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 7, 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 6 is sorted by the EV6-EV67 column to highlight the widest differences between

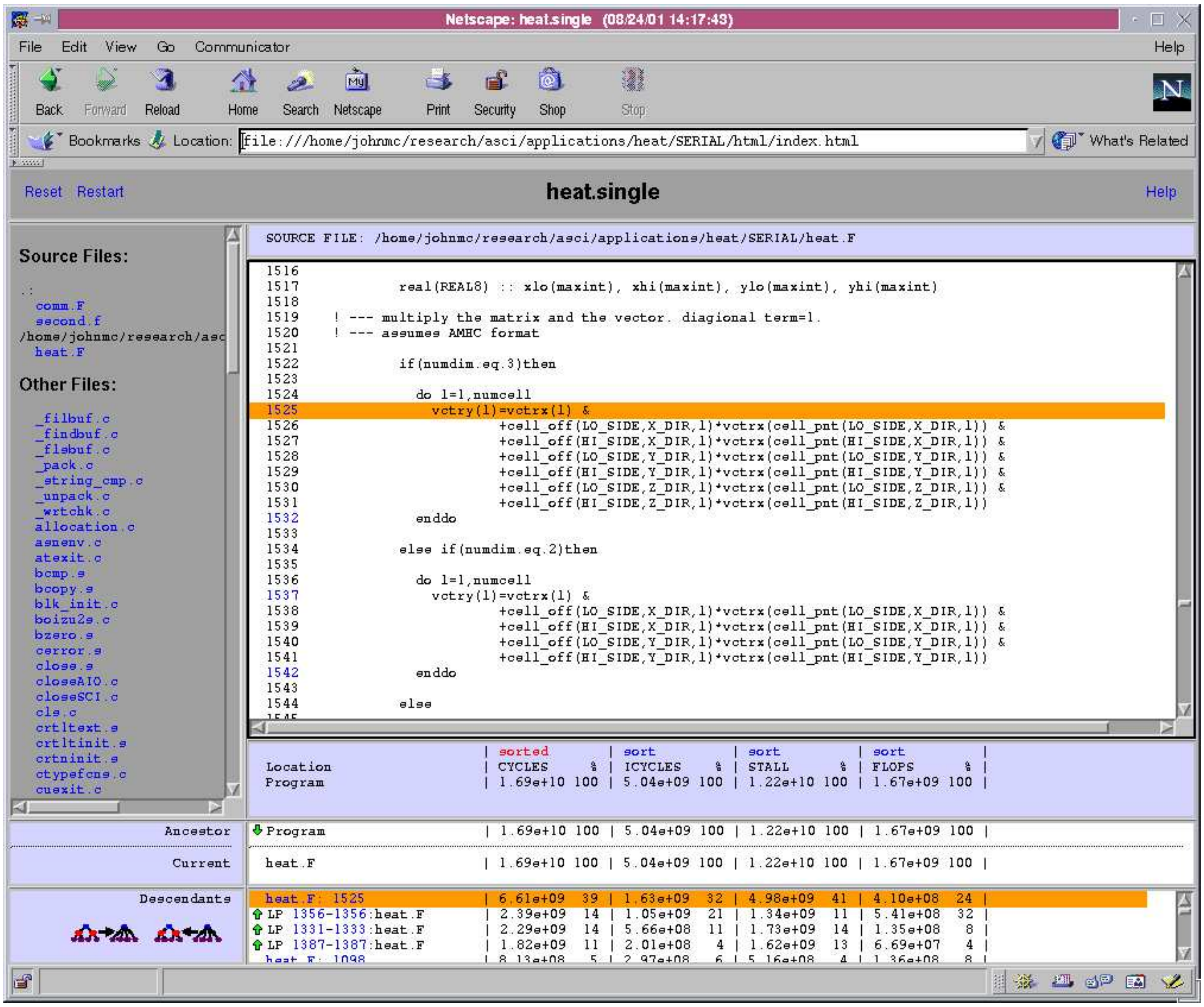

Figure 5: Using HPCView to display both measured and computed metrics.

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. For the whole program, the ratios swing back again to 1.99 and 1.38. While some part of these difference 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. This example illustrates the importance of being able to examine and compare performance measurements at sevel levels of aggregation.

4.2 Discussion

The ability to assemble data from multiple sources for analysis 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 examples 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

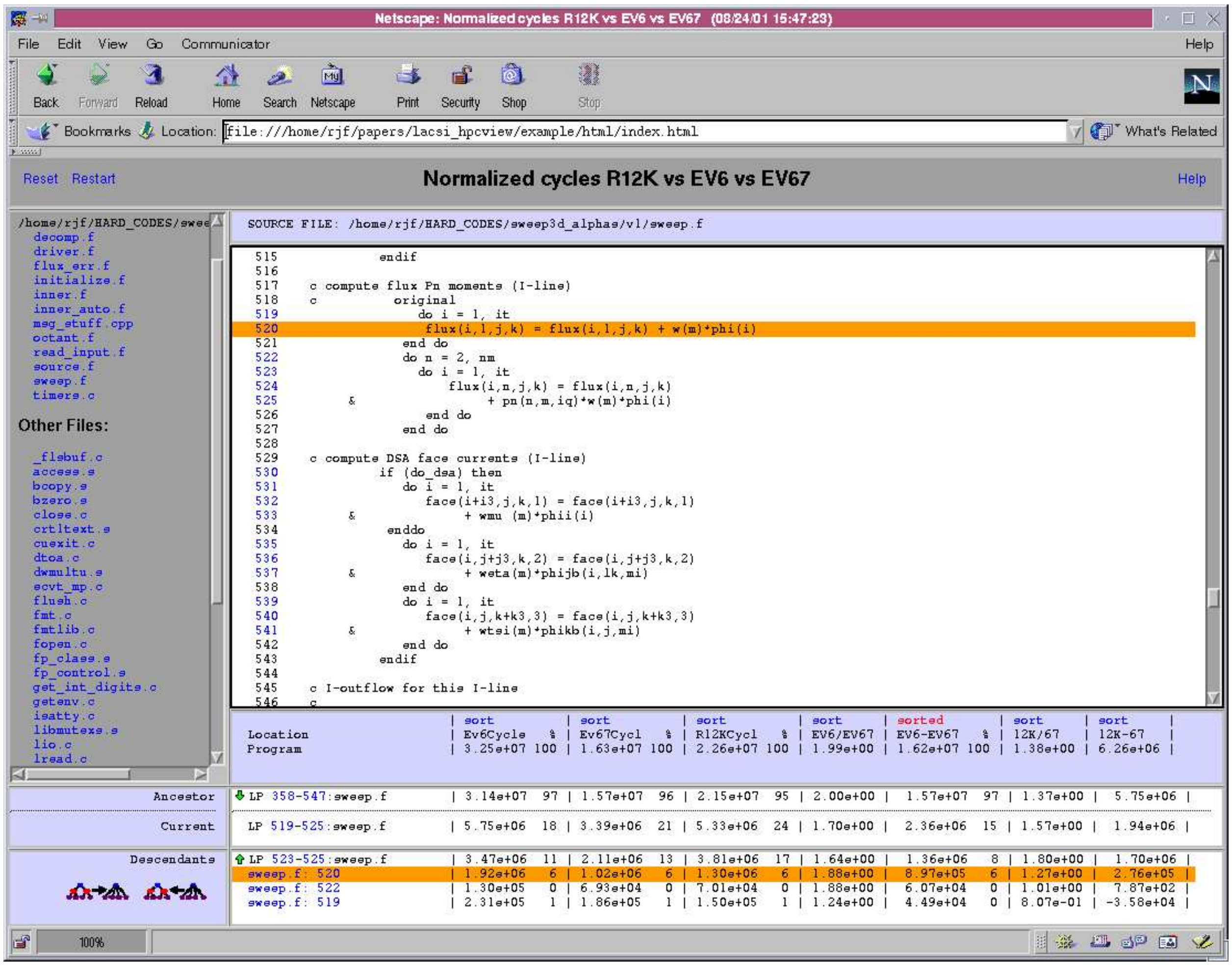

Figure 6: Comparison of cycle counts normalized to microseconds on three architectures.

performance between different architectures and between different implementations of the same architecture.

5 Related Work

Performance monitoring can be made more efficient by making use of special hardware features beyond simple event counting. For instance, the experimental FLASH multiprocessor had a programmable cache controller. *FlashPoint* [16], 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.

The idea of computing the number of cycles lost due to architecture and system overheads has appeared several times in the literature. *MTOOL* [6] 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 data objects involved was often difficult to determine from *MTOOL*'s output [6].

Going beyond attributing cycles "lost" to the memory hierarchy, *lost cycles analysis* [15] classified all of the sources of overhead (waiting time) that might be encountered by a parallel program. The Carnival tool

```
<HPCVIEW>
<TITLE name="Normalized cycles R12K vs EV6 vs EV67 " />
<PATH name="." />
<STRUCTURE name= "sweep.bloop" />
<METRIC name="a" displayName="Ev6Cycles" display="false">
 <FILE name="sweepsing.ev6cy.pxml" /> </METRIC>
<METRIC name="b" displayName="Ev67Cycles" display="false">
 <FILE name="sweep.ev67cy.pxml" /> </METRIC>
<METRIC name="c" displayName="R12KCycles" display="false">
 <FILE name="../sgiv1/mapy.fcy_hwc.pxml" /> </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>
</HPCVIEW>
```

Figure 7: A *HPCView* configuration file used to compute the derived metrics displayed in Figure 6.

set [4] extended this into "waiting time analysis". It provided a visualization tool that is similar in spirit to the *SvPablo* display, 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 can be configured to count clock cycles or selected performance-related events. Profiling using these counters is facilitated by allowing counter overflows to 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* [19] 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* [11] also displays a single performance metric by annotating each line with a count.

Sun Microsystems provides a rich GUI interface to performance measurement and display tools using the Workshop programming development environment[17] and the `dbx` debugger. Under Solaris8, these tools were extended to take samples, including the full callstack, on performance counter overflow. While the Sun tools provide a generic set of basic profiling and analysis tools, the advanced features are largely orthogonal to the concerns addressed by HPCView. The Sun tools can display multiple metrics, but there is no provision for computing derived metrics. While data can be aggregated inclusively, and exclusively up the call chain, there is no way of aggregating metrics at scopes below the procedure level.

Both the Sun analyzer tool and the programs for displaying information from Compaq's Dynamic Continuous Profiling Infrastructure have options for displaying performance histogram data in a listing of disassembled object code annotated with source code. While this makes it possible to analyze highly-optimized code where a single source line may map into multiple instances in the executable, it is a difficult and time consuming exercise. Adve *et al.* [1] demonstrated a performance browser that uses compiler-derived mapping information to interactively correlate HPF source code with the compiler's parallelized, optimized F77+MPI output, which is instrumented with SvPablo (See below.). This combination can present MPI performance data in terms of both the transformed and original code. Vendor compilers are increasingly able to generate annotated listings that attempt to explain both what optimizations have been performed and what aspects of the program inhibit optimization. To our knowledge, performance data has not been integrated with this information for presentation.

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 [14]. 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 displays to facilitate top-down analysis.

6 Concluding Remarks

In this paper we described some of the design issues and lessons learned in the construction and use of HPCView, a performance analysis toolkit intended specifically to increase programmer productivity in performance engineering by being easier to use, while providing powerful more analytic capabilities than existing tools.

As described in previous sections, we have used *HPCView* on several whole applications. Notably, 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 enabled us to quickly identify significant opportunities for performance improvement. For large codes, however, the HTML database size grows large when many metrics are measured or computed. Currently, we precompute static HTML for the entire set of potential displays. For example, instead of dynamically sorting the performance metric panes, we write a separate copy of the data for each sort order. These separate copies are written for the data in 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[4]. These databases correlate performance metrics to all of the application sources. To reduce the size of performance databases, we are prototyping a stand-alone performance browser written in Java that creates views on demand.

Acknowledgments

Monika Mevencamp and Xuesong Yuan were the principal implementers of the first version of HPCView. 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

[4]The amount of wasted and redundant space is illustrated by the fact that the `gzip`'ed `tar` file is less than 1.5 megabytes, most of which represents compressed source code.

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.

References

[1] V. Adve, J.-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[2] 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.

[3] Kirk W. Cameron, Yong Luo, and Janes Scharmeier. Instruction-levle microprocessor modeling of scientific applications. In *ISHPC 1999*, pages 29 – 40, Japan, May 1999.

[4] Carnival Web Site. `http://www.cs.rochester.edu/u/leblanc/prediction.html`.

[5] 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.

[6] 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.

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

[8] Sweep3D Benchmark Code. DOE Accelerated Strategic Computing Initiative. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`.

[9] 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.

[10] K. Kennedy J. Mellor-Crummey, D. Whalley. Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the ACM International Conference on Supercomputing '94*, pages 425–433, June 1999.

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

[12] T.Y. Johnston and R. H. Johnson. Program performance measurement. Technical Report SLAC User Note 33, Rev. 1, SLAC, Stanford University, California, 1970.

[13] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13(3):313–322, 1973.

[14] 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.

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

[16] 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.

[17] Sun Microsystems. *Analyzing Program Performance With Sun WorkShop*, 2001. http://docs.sun.com/htmlcoll/coll.36.7/iso-8859-1/SWKSHPPERF/AnalyzingTOC.html.

[18] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

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