A Sample-Driven Call Stack Profiler

Nathan Froyd, John Mellor-Crummey, and Rob Fowler

July 15, 2004

Abstract

Call graph profiling reports measurements of resource utilization along with information about the calling context in which the resources were consumed. We present the design of a novel profiler that measures resource utilization and its associated calling context using a stack sampling technique. Our scheme has a novel combination of features and mechanisms. First, it requires no compiler support or instrumentation, either of source or binary code. Second, it works on heavily optimized code and on complex, multi-module applications. Third, it uses sampling rather than tracing to build a context tree, collect histogram data, and to characterize calling patterns. Fourth, the data structures and algorithms are efficient enough to construct the complete tree exposed in the sampling process. We describe an implementation for the Alpha/Tru64 platform and present experimental measurements that compare this implementation with the standard call graph profiler provided on Tru64, hiprof. We show results from a variety of programs in several languages indicating that our profiler operates with modest overhead. Our experiments show that the profiling overhead of our technique is nearly a factor of 55 lower than that of hiprof when profiling a call-intensive recursive program.

1 Introduction

As the size and complexity of programs increase, identifying resource utilization hot spots within the code becomes more and more difficult. Automatically measuring resource utilization, known as profiling, is often far superior to the intuition of even expert programmers for identifying hot spots. Many tools have been developed to pinpoint performance problems and they can be categorized by the kind of profile which they collect. The first category is known as flat profilers [2, 24]. Flat profilers collect a program counter-based histogram of the performance metric of interest during the program’s execution. Using the object-to-source mapping information generated by the compiler, the profiling tool then associates the data with the source program and presents it in a format suitable for analysis. Flat profilers offer several advantages: they are simple to implement; they add a small amount of overhead; and they generally do not require any preparation on the part of the programmer.
For programs in which procedures are called in only one context or in which procedure behavior is context-independent, flat profiles are often sufficient. For modern, modular codes, however, it is often necessary to partition the costs incurred in a procedure according to the contexts in which that procedure is called, especially for workhorse routines in standard libraries that may be called in many ways. Since the system and application can both contribute multiple layers of module and library, instrumenting any single library is usually insufficient. For example, a flat profiler may report that large amounts of time were spent in a low-level system routine such as `bcopy`, or a synchronization primitive. Attributing the costs from such routines to their call sites, however, is far more illuminating for determining where the performance problems really lie. To do this in a general way requires capturing extensive context information.

A call graph is a directed graph in which the nodes are in one-to-one correspondence with the procedures in a program, and the edges represent the caller-callee relation, either potential (because the procedure call code exists in the source), or actual (because it was actually invoked in a particular run of the program).

There are many definitions of Call or Context Tree in the literature. Starting with the nomenclature of Ammons, Ball, and Larus[1], a Dynamic Call Tree is a rooted tree in which each node represents a unique invocation of a procedure, and each edge represents the caller-callee relation. The size of a dynamic call tree is proportional to the number of calls made in an execution. A Context Tree is a rooted tree in which the root represents the initial invocation of the main procedure; each edge represents the aggregate caller-callee relation between the parent node and the child node; each node represents the aggregate of the set of invocations of a procedure from each “unique” calling context; and, thus, the path from the root to each node represents the call chain to that node. By changing the test for uniqueness of the calling context one can generate compact, but less precise, approximations of the Context Tree. For example, uniqueness could be determined by matching only the last \( k \) call frames[23], or by collapsing recursive calls[1].

Context-based profilers work with one or the other of these structures by annotating nodes and/or edges with performance metric data.

Some context-based profilers can be viewed as exhaustive because such they update calling context information at every call and return. Such profilers are implemented using compiler-based instrumentation inserted procedure entries to discover nodes and edges in the call graph or tree [8, 5, 20]. Some effort must be expended to ensure that the data returned from this sort of approach is accurate [17, 22]. Requiring source code and compiler support for performance analysis of applications is a severe restriction, especially for real applications which contain modules written in several programming languages (Fortran dialects, C, C++, Java, Python), and use libraries requiring extremely long re-compilation time, even when source code is available. Thus, binary rewriting tools [18, 14, 21] have helped free the instrumentation phase from dependence on the compiler, but work is still done on every call and return.

Executing instrumentation code on every call and return can drastically increase the amount of time required and there is no easy way to control this cost. This problem is addressed
in a second category of call graph profilers [3, 10, 23, 4], which record performance and context information on a periodic basis, usually through sampling combined with call stack analysis. Because sampling is used the resulting structure approximates the context tree in two ways: rarely called paths will not be captured and the accuracy of the performance data is limited by the sampling granularity. By taking enough samples, a sufficiently accurate approximation to a context tree can be constructed and those aspects of program behavior that have a significant effect on overall performance will be represented accurately. The tradeoff between execution overhead and accuracy can be controlled directly by controlling sampling frequency. Furthermore, since accuracy over an interval is primarily dependent on the absolute number of samples collected in that interval, one can choose to gather the data by doing intensive, high overhead sampling on a benchmarking run, or to do minimally invasive sampling on a long production run.

Our profiler combines the flexibility of the sampling-based approach on unmodified binaries with an efficient implementation that makes it feasible to collect the full, sampled context tree. Furthermore, we use an innovative method for capturing calling pattern information. The system is designed to work on full scale applications that use dynamically loaded external libraries, threading, and exception handling.

We describe the principles of our approach in section 2 and follow up with an overview of our implementation of these principles on the Alpha/Tru64 platform in section 3. We present some preliminary experimental results that illustrate the overhead of our profiler and compare it to HP’s hiprof tool in section 4. We conclude by discussing future directions.

2 An Abstract and Portable Design

Immediately before an unmodified application program is launched, an environment variable is set to cause a profiling library to be pre-loaded before the application and its shared libraries are loaded. Thus, the initialization routine of this library initiates profiling before the application is even loaded. After the application finishes execution, a finalization routine of this library is the last thing to run. It halts the profiling and writes the collected profile data to a file.

Because the profiler lives in the address space of the application, it has full access to data and code, and we avoid context switching overhead.

2.1 Collecting Samples

Sampling events can be supplied by any of several mechanisms. The three requirements are that hardware can generate a trap at the end of an interval in which a known amount of an interesting performance metric has been consumed, that the program counter when the trap occurs points to an instruction that “consumes” the metric, and that the operating system deliver this information to the application as a POSIX signal. The implementation
we describe below uses a SIGPROF signal generated by an interval timer. Other systems can use overflow traps from hardware performance counters to generate sampling events. When an event occurs, the “cost” associated with the event is charged to the instruction in the active procedure at which the trap occurred and, implicitly, to the entire current chain of call sites.

The data samples are accumulated in a straightforward and complete representation of the context tree seen thus far. In this representation, each edge is associated with a label, the return address of a call site in the application and the application’s associated libraries. The implicit label of each node is a path label, the concatenation of the labels of all of the edges from the root to that node.

In principle, a new sample could be added to the context tree by first extracting the path label location associated with the sample location by unwinding the current call stack, recording all the return addresses along the way. The correct node in the context tree could then be found by following a labeled path down from the root, allocating new edges and nodes if necessary. This would be comparatively expensive because of the number of instructions and the number of distinct memory locations that need to be touched.

To reduce the cost, we use two techniques to limit the amount of memory that needs to be touched.

First, we memoize the chain of return addresses that were found when the last sample was recorded. Associated with each element in this stack cache is a pointer to the context tree node labeled with the prefix of the return addresses in the memoized chain. To record a new event, the profiler first, in principle, creates the memoized chain of return addresses for the current event. The two chains are then merged by walking down from the root to the last place where the two lists agree. The pointer directly into the context tree is then followed and suffix of the chain is processed in the tree. As this is done, pointers to the tree nodes are added to the memoized chain for the current event.

Using this scheme, the amount of memory that needs to be touched to verify that the call chain has not changed is equal to a traversal of the current call chain plus structures twice the length of the return address chain. In most scientific codes, these latter structures occupy only a few cache lines. This is a big improvement over the cost of traversing the context tree, but because the entire call stack must still be traversed in order to check that nothing has changed near the root, it is still too expensive.

2.2 Limiting Stack Traversal

The full traversal of the call stack at each sampling event is necessary to verify the path label of current stack. If the profiler itself does not mutate the stack, this is the only way to guarantee that between events there was not a sequence of calls and returns that constructed a call chain with a similar suffix to what was seen previously. In contrast, Arnold and Sweeney[3] used a modified Java compiler to insert sentinel bits in each call frame that are cleared by the profiler each time it examines the frame. Thus, because the profiler can distinguish between
Figure 1: Operation of the profiler: (a) the program is running normally; (b) a sample event occurs and the trampoline is inserted; (c) readcfg returns to the trampoline; (d) the program resumes normal execution after the trampoline has moved itself up the call chain; (e) barrier is called and the trampoline is not moved; (f) a sample event occurs and the trampoline is moved to catch the return from barrier.

frames it has examined and those it hasn’t, it only needs to walk back up the stack until it finds a frame it has seen before.

Since it is impractical to recompile with a modified set of compilers all the parts of a full scale application and all of the libraries it uses, we needed to use another, purely runtime approach to limit the stack search.

The method we use is for the profiler to modify the stack by splicing a trampoline into the stack each time a sample is recorded. To do this, the sampler replaces the return address in the active procedure frame with the address of our trampoline and saves the actual return address in a structure parallel to the stack cache for use by the trampoline. It also stores a pointer to the original location of the return address. When the current procedure tries to return, it instead transfers control to the trampoline. The trampoline re-installs itself one frame higher in the call chain, and then transfers control to the the saved return address. The trampoline also (See section 2.3.) has access to the context tree and it has the opportunity to augment the profiling information.

With a trampoline mechanism in place, the trampoline itself serves as the sentinel that separates the part of the call stack that has been modified since the previous sample from the unchanged part. Therefore, on a sampling event, the call stack only needs to be traversed until the trampoline is found. The portion of the stack above that point has not been modified since the last event.
When the next sample is taken, the current trampoline is spliced out, and a new one is inserted in the active procedure. (This is the reason for recording where the original return address was kept.) Thus, each thread of control in a program has at most one trampoline active at a time. Figure 1 illustrates the movement of the trampoline in a profiled program.

2.3 Using the Trampoline to Expose Calling Patterns

Although the sampled events measure where costs were accumulated in the context tree, they do not tell the whole story. For example, it may reveal that the application spent a significant fraction of its time in procedure $f$ when called along a particular call chain. It does not, however, offer any insight as to whether this is because each invocation of $f$ is particularly expensive or $f$ was called a very large number of times.

To get a handle on the calling patterns, we use the trampoline mechanism to augment the sample count information. Each time the trampoline is invoked, it increments an “sampled call sites counter” for the context that just returned. The result is to accumulate counts on call chain edges that existed when a sample was taken, but that were removed before the next sample occurred.

With some minor exceptions, the sampled call sites counter is equivalent to sampling “new call edges” as was done by Whaley[23] and by Arnold and Sweeney[3]. The latter found that it was an unsuitable measure for their goal of approximating procedure call counts. These measures are, however, much better suited to our purpose of providing an indication of the range of call stack activity between sampled events.

2.4 Handling Important Program Features

The abstract design presented above is sufficient if one is only interested in profiling simple examples in which there is a single stack in memory that is modified only by procedure calls and returns. Real applications, however, are extremely complex, and even a portable design needs to deal with these complexities.

2.4.1 Dynamic Loading

Because shared libraries can be loaded and unloaded during execution, a particular address may refer to several different functions during a run of the program. Mapping program counters in the collected profile to their associated functions requires tracking the shared libraries used by the program. We call the list of libraries and their load information when a particular sample is taken the sample’s epoch. In each epoch, the sampled addresses are unambiguous. A new epoch is created when each library is loaded or unloaded. When a call stack sample is taken, the sampler must check to see whether the epoch has changed since the last sample taken. If the two match, then the sample-taking may proceed as normal. Otherwise, a new sampling tree is created and bound to the current epoch.
A new epoch can also be initiated to start collection of a new profile when the program enters a new phase.

2.4.2 setjmp and longjmp

The normal sequence of procedure calls and returns is often disrupted in real programs. A prime example of this is an exception handler in which control is not returned to the procedure which throws an exception. Instead, control passes to a saved context further up the call stack and the procedure throwing the exception does not return to its immediate caller. In POSIX systems, the standard C library functions setjmp and longjmp implement this functionality. As our technique is intimately involved with the call stack, it must handle the unique conditions posed by these functions.

The key insight is that setjmp and longjmp deal with execution contexts, which consist of the values of the processor registers, the instruction pointer, and a few miscellaneous details. In our profiler, the execution context may have been augmented with the trampoline and the saved return address. These must be preserved along with the rest of the “normal” execution context which setjmp and longjmp use.

We therefore modified setjmp so that, each time it is called, the execution context is augmented with a small structure containing the trampoline information and a slot for a value of a chosen processor register, \( r \). A pointer to this structure is then stored in the location in the context which normally holds the saved value of \( r \). If a particular execution context is restored via longjmp, the stored value of \( r \) is used to locate and extract the trampoline information and these modifications to the saved context undone by restoring \( r \).

2.4.3 Multithreading

As long as the source of sampled events can deal with threading, modifying the profiler for threaded programs is straightforward; each thread receives its own copy of profiling state as well as its own trampoline. For performance metrics which are thread-local, e.g., resources consumed within the thread, the sampling portion of our profiler requires no communication between threads. For performance metrics that are program-wide, e.g., wall clock time, minimal coordination between threads is needed to initiate the sampling process.

3 Alpha/Tru64 Implementation

Getting everything to work correctly requires attention to detail, especially when profiling heavily optimized code. In this section, we describe some of the specific, non-portable implementation techniques used in our Alpha/Tru64 implementation.
3.1 Timer-Based Sampling

Samples are taken by using Unix signals and the interval timers provided by the Unix kernel. Each process is given three interval timers; we use the ITIMER_PROF timer, which decrements during user mode execution and when the kernel is running on behalf of the process. A signal is sent to the process every \( n \) microseconds (user-configurable) and we install a signal handler during the initialization of our profiling library to catch this signal.

While the Alpha microprocessor contains hardware to perform sample-based profiling using programmable performance counters, there is no user-level notification of when a performance counter overflows. Thus we cannot use the programmable performance counters on Alpha/Tru64 as a sample source. On other platforms, such as IA-64/Linux, where counter overflow is optionally exposed to the program [6], we could exploit hardware performance counters as a sample source. Alternatively, we could monitor other events, such as the number of bytes allocated, with some minor modifications to our existing scheme.

3.2 Unwinding the Stack

Obtaining call stack samples is a simple matter of knowing the size of each procedure’s stack frame and where the return address is stored. The calling standard for Tru64 requires this information about functions, collectively known as “runtime procedure descriptors” (RPDs), to be included with the compiled object code. A convenient interface for accessing RPDs and walking the stack is libexc.so, a library intended for implementing structured exception handling in languages like C++. We use this library for unwinding the stack and determining where to insert our trampoline.

When our signal handler is called, we determine whether or not it is “safe” to collect a call stack sample. For instance, while the runtime loader is executing, walking the call stack is impossible since the functions in the runtime loader do not contain the information necessary for unwinding the call stack. If the context in which the signal handler was called is not “safe”, we increment an “unsafe samples” counter and then return from the signal handler. Otherwise, a call stack sample for the signal context is collected. There are also several places in the profiler where either obtaining a call stack sample or inserting the trampoline is not possible; these contexts are detected upon entrance to the signal handler as “unsafe”. The number of “unsafe samples” is reported as the number of counts missed during the execution of the program.

3.3 Overriding Library Calls

Since our profiler is running in the same address space as the process being profiled, there are several ways in which the process can shut off the profiler, either by accident or on purpose. For example, the application might try to install its own signal handler for SIGPROF. To prevent such actions from occurring, we override several functions in the standard C
library with functions of the same name in our library. Dynamic linking ensures that our
versions are used instead of those in the C library. When the profiling library is initialized,
function pointers to the real versions residing in the C library are captured. Our alternate
implementations can then apply any necessary preprocessing to the arguments and then
call the real functions. For instance, our version of setitimer ensures that the profiled
program is not setting the ITIMER_PROF timer, which could disturb our measurements. Other
functions, such as dlopen, setjmp, and longjmp are overridden to provide the necessary
hooks to provide the functionality described in Section 2.4, above.

3.4 Inserting the Trampoline

To insert our trampoline, we must be able to determine where the return address for the
current procedure is stored. RPDs come in two different flavors: one for procedures which
build a stack frame and another for procedures which store the caller’s context in registers
(“register frame procedures”). In the first case, the RPD indicates the distance from the
bottom of the stack for the return address storage area. Finding the return address is then
simple arithmetic with some extra handling in case the return address has not yet been
stored there, e.g., if the signal occurs during the prologue of a procedure. For register frame
procedures, the RPD stores the number of the processor register which contains the return
address; the saved content of this register is accessible in the signal context passed to the
handler.

One issue with our trampoline-based scheme involves link-time optimization. If f is a func-
tion that is only called by another function g, but f and g are not in the same compilation
unit (and therefore f cannot be inlined), then the Tru64 linker can optimize f to return
through a branch instruction. Our trampoline depends on functions returning through an
indirect jump instruction to the stored return address, so f’s return would not be detected.

When a trampoline is installed to catch f’s return, the location of the return address for g
is actually the location modified, since the return address for f is implicitly encoded in the
direct branch instruction and not modifiable. When g returns, the trampoline will notice
that the cached call stack contains an “extra” node – the node for f which was never removed
– and pop that node in addition to the node for g.

3.5 Multiple Threads and Signals

When multiple threads are running, each thread independently maintains its own trampoline
and cached call stack. However, only the call stack for the thread which receives the SIGPROF
signal from the operating system is sampled, rather than notifying each thread to take a
sample. Assuming that the thread which is notified of the signal was actually running
on the processor, this scheme will produce accurate results for a single-processor system.
On a multi-processor system, the situation is more complex, as there is not a method for
discovering the thread currently executing on each processor and sampling those threads
Discovering thread creation is accomplished by overriding pthread_create. Our profiler supports monitoring any thread created through pthread_create, whether it is a user-level thread or kernel-level thread. Our version of pthread_create creates a small structure containing the user-passed parameters for the thread, passing the real pthread_create function this structure and an alternative initialization function. This alternate function initializes the profiling state in the context of the created thread rather than the parent thread and thus minimizes thread-to-thread communication.

4 Experimental Results

This section presents experimental results obtained by using our profiler for several synthetic and real-world programs. We also compare our overhead to that of programs instrumented with HP’s hiprof tool. Programs were chosen to represent the kinds of programs we anticipate our profiler being used to analyze. When possible, we compiled the programs with high levels of optimization using the system compiler.

We are currently unable to provide results for multi-threaded programs. The version of Tru64 which we used for our testing, 5.1A, contains problems in its system libraries which do not permit the profiler to run correctly for multi-threaded programs. In addition, several system libraries contain functions which violate the calling conventions. These calling convention violations have been worked around, but the workarounds require examining significant portions of assembly code during each stack unwind. These bugs have been fixed in the most recent revision of Tru64, but we do not yet have access to systems with recent revisions installed. Performance on recent revisions should be somewhat better than the figures which we present here.

We profiled three programs:

1. A home-grown sort benchmark, which allocates a multi-megabyte vector of random numbers and sorts the vector with the C++ STL’s sort call. A vector of one hundred million unsigned 32-bit integers was used along with the Mersenne Twister pseudo-random number generator [15] for generating the random numbers.

2. The ASCI benchmark sweep3d [12], which solves a 3D neutron transport problem. sweep3d is written almost entirely in Fortran 77. We tested the serial version of sweep3d with the supplied 150x150 problem. sweep3d’s built-in timers were used to provide timing statistics.

3. The Parallel Ocean Program (POP) [13, 11], an ocean circulation model simulation developed at Los Alamos National Labs written in Fortran 90. We tested the serial configuration of the code and used the default parameter file for our tests. POP contains built-in timing routines; we used the final value of the TOTAL timer for our measurements.
Ten runs of each program were done with both our tool, \texttt{csprof}, and HP’s \texttt{hiprof}, in addition to ten unprofiled runs (thirty runs in total). The results were then averaged for each tool. In table 4, the “unprofiled” column indicates the time taken for the uninstrumented program and the “csprof” and “hiprof” columns report the results for each profiling tool. Execution times are in seconds; the “time/sample” column is recorded in milliseconds. All tests were run on an unloaded four processor 667MHz Alpha EV67 machine with 2GB of main memory running Tru64 5.1A.

\texttt{hiprof} was instructed to instrument all associated libraries for each application and sampled the program counter every 1000 microseconds, whereas \texttt{csprof} sampled the call stack every 5000 microseconds. We could not find a way to alter \texttt{hiprof}’s sampling frequency and raising \texttt{csprof}’s sampling rate did not have any noticeable effect on the number of samples collected.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>unprofiled</th>
<th>hiprof</th>
<th>csprof</th>
<th>csprof samples</th>
<th>time/sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>39.46</td>
<td>67.42</td>
<td>39.97</td>
<td>7752</td>
<td>0.066</td>
</tr>
<tr>
<td>sweep3d</td>
<td>409.23</td>
<td>413.40</td>
<td>417.82</td>
<td>71591</td>
<td>0.120</td>
</tr>
<tr>
<td>pop</td>
<td>109.99</td>
<td>120.86</td>
<td>119.54</td>
<td>20362</td>
<td>0.469</td>
</tr>
</tbody>
</table>

On Tru64, C++’s \texttt{sort} call is implemented with a recursive quick sort augmented with a final pass of insertion sort. \texttt{sort} therefore has a very large number of calls and returns, penalizing \texttt{hiprof}’s instrumentation of calls and returns. \texttt{csprof} has a very low overhead on \texttt{sort}, the reasons for which will be explained shortly.

\texttt{sweep3d}’s execution time is dominated by one procedure, \texttt{sweep}, which accounts for 95% of its running time. The number of calls and returns is therefore very low, accounting for \texttt{hiprof}’s low overhead for this particular application. \texttt{csprof}, however, has an overhead primarily dependent on the sampling rate, and turns in a slightly larger overhead.

\texttt{pop} falls somewhere between the high call/return activity of \texttt{sort} and the extremely low activity of \texttt{sweep3d}. Nearly 60% of \texttt{pop}’s time is spent in a single procedure, but it also contains a significant number of other procedures which are called as well. \texttt{hiprof} and \texttt{csprof} incur approximately equal overhead for this program.

We were puzzled by \texttt{csprof} incurring virtually the same percentage of overhead on \texttt{sort} and \texttt{sweep3d}, despite their radically different calling activity. Since we cannot profile anything that happens in signal handler or in the trampoline with our profiler (\texttt{SIGPROF} is blocked during the execution of both), we turned to DCPI [2] and HPCToolkit [16] to analyze this problem. Profiling \texttt{pop} showed the bulk of time spent in the profiler was to insert our collected samples into the context calling tree. Our current design stores the children of each node in the tree as a doubly-linked list; finding the node which matches a particular program counter is therefore an expensive pointer-chasing search through memory.

This explains the overheads observed for both \texttt{sort} and \texttt{sweep3d} as well: nodes in \texttt{sort}’s and \texttt{sweep3d}’s calling context tree have few children and so inserting new samples is inexpensive. In \texttt{pop}’s case, the calling context tree is quite wide and having more children is common,
which makes inserting new samples expensive due to the cache-unfriendly storage of child nodes. We plan on examining the data structures used for our calling context tree and tuning them for cache usage, which will decrease our overhead 30% or more.

5 Current Directions

One problem for which there is currently no adequate solution is the presentation of data gathered by context-based profilers. Ideally, a tool for viewing the collected data would allow the user to view the overall structure of the call graph in some fashion and then iteratively narrow or widen the scope of the view.

An obvious idea with which we have experimented is to draw a graph where functions are nodes and edges represent calls between functions. There are several packages available which, when supplied with specifications for the nodes and the edges, will automatically draw a “pretty” graph for a human viewer [7, 19]. We have found, however, that for moderate-sized programs this method becomes unwieldy and unhelpful. In addition, if the user wishes to focus on a subset of the nodes in the graph, the underlying graph layout package will probably have to recompute the layout of the new graph. Doing this has a high probability of rearranging nodes and edges, confusing the user in the process.

The most compelling idea we have seen in this area comes from Hall’s work on call path refinement profiles [9]. In Hall’s scheme, focusing on particular paths in the graph is accomplished primarily by call path refinements, which are filters indicating for which paths performance data should be computed and displayed. For example, when presented with the call tree of a large scientific code, the user might request that only the paths which contain calls to synchronization routines be displayed. Next, the user might further refine the display to only those paths containing calls to the differential equation solver be displayed. This new filter is only applied to those paths containing calls to synchronization routines, allowing the user to effectively zoom in on a very specific area of the program. Hall also suggests being able to aggregate several routines together so that they would display as a single routine while examining the profile; this behavior would be useful for focusing on the implementation of a data structure (e.g., hash tables) as a data structure rather than on disparate routines.

HPCToolkit (formerly called HPCView [16]) is a collection of tools for correlating multiple flat performance data profiles with source code. Although the input data consist of flat profiles, for analysis and presentation HPCToolkit aggregates the data hierarchically using the static structure of the program (load modules, files, procedures, loop nests, and statements). HPCViewer is the performance browser component of HPCToolkit. It encourages a top-down approach to browsing the data by providing mechanisms for manipulating the programmer/analyst’s view of the hierarchy as a tree while displaying a performance data table for those elements of the tree that are currently visible. This has proven to be a very productive way of presenting the data because it allows the user to start with the whole program and to quickly explore the significant parts of the program by adjusting the refinement
of the static tree.

Our current plans are to extend the HPCViewer approach by augmenting the view navigated through the static program structure tree by adding a similar hierarchical view of the dynamic context tree, including Hall-style refinements, and by providing links between the two views.

We are also working on interacting properly with the multi-threaded runtime of Tru64 to correctly profile multi-threaded programs. Cooperating with other runtime conditions (e.g. C++ exceptions) is an active area of implementation.

Using DCPI and HPCTookkit, we are currently examining the detailed performance of our implementation and expect to report a significant improvement in performance in the next revision of this paper.

6 Conclusion

Understanding the performance of large scale modular codes requires knowledge of the calling context of performance-critical routines. We have presented the design of a profiler which collects the dynamic call tree of a program through sampling as well as providing edge counts for further assistance in locating performance problems. Our profiler handles standard program features and requires no compiler support nor binary instrumentation. An implementation for Alpha/Tru64 was demonstrated and its overhead was shown to be no worse and sometimes much better than a standard instrumentation-based profiler.

7 References

References


