RICE UNIVERSITY

Efficient Call Path Profiles on Unmodified, Optimized Code

by

Nathan Froyd

A Thesis Submitted
in Partial Fulfillment of the Requirements for the Degree

Master of Science

APPROVED, THESIS COMMITTEE:

John Mellor-Crummey, Associate Professor, Chair
Computer Science

Ken Kennedy, Ann and John Doerr Professor of Computational Engineering
Computer Science

Keith Cooper, Professor
Computer Science

Alan Cox, Associate Professor
Computer Science

Houston, Texas
April, 2005
INFORMATION TO USERS

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

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

Umi Microform 1425828

Copyright 2005 by ProQuest Information and Learning Company.

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

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346
Efficient Call Path Profiles on Unmodified, Optimized Code

Nathan Froyd

Abstract

Identifying performance bottlenecks and their associated calling contexts is critical for tuning high-performance applications. This thesis presents a new approach to measuring resource utilization and its calling context. Previous instrumentation-based approaches for reporting calling context introduce overhead proportional to the number of function calls performed. We describe a new design for a call path profiler based on stack sampling. Our design enables profiling of unmodified binaries, provides low and controllable overhead, and accurately attributes context-dependent costs of calls. We use a special trampoline function that improves the efficiency of stack sampling and enables the association of unique invocation counts with sampled call sites. We evaluate a Tru64/Alpha implementation of our design and show that on call-intensive codes, the overhead of our approach is over two orders of magnitude lower than the overhead of an instrumentation-based approach, with comparable overhead on other codes.
Acknowledgments

I am indebted to my advisor, John Mellor-Crummey, for the numerous suggestions he offered to improve the presentation of this thesis as well as his encouragements during the course of the thesis research. Thanks are also due to my committee members Ken Kennedy, Keith Cooper, and Alan Cox; answers to their questions have found their way into this thesis and the thesis is the better for it.

Jan Hewitt reviewed several drafts of this manuscript and provided many helpful suggestions. The thesis-writing course she allowed me to be a part of proved invaluable in focusing on what needed to be done.

Nathan Tallent wrote the “original” cprof for the Linux/Itanium platform; I am thankful for all of the bug-free code he provided for me. Much of the code in the implementation for this thesis still bears his mark. All remaining bugs are my own.

My family has been a constant source of encouragement during this work, for which I am grateful.

Finally, innumerable thanks are due to my wife, who has patiently listened to many explanations of the ideas presented in this work and has boosted my spirits many times during my time here at Rice.
Contents

Abstract i
List of Tables v

1 Introduction 1

2 Related work 6
  2.1 Data structures ........................................... 6
  2.2 Instrumentation-based profilers .......................... 7
  2.3 Sampling-based profilers ................................. 12

3 High-level profiler design 16
  3.1 How to collect samples ................................. 16
    3.1.1 Sample events ...................................... 17
    3.1.2 Taking a sample ................................... 19
  3.2 Efficient unwinding with a trampoline ................. 21
    3.2.1 Marking an activation record with a trampoline .. 24
    3.2.2 Sampling with a trampoline ...................... 34
  3.3 Coping with program features .......................... 34
    3.3.1 Unsafe functions .................................. 35
    3.3.2 Dynamic linking and loading ...................... 36
    3.3.3 Optimized calls .................................. 38
    3.3.4 Frame pointer canonicalization ................... 42
    3.3.5 Overriding library calls .......................... 43
    3.3.6 Non-local exits ................................... 47
4 Alpha/Tru64 implementation 53
  4.1 Sampling and unwinding details .............................. 53
  4.1.1 Unsafe function detection .................................. 55
  4.2 Implementation difficulties ................................. 56
  4.2.1 Marking activation records ............................... 57
  4.2.2 Calling convention violations ............................ 59
  4.2.3 Procedure descriptor bugs ................................ 62

5 Experimental results 64
  5.1 General setup .................................................. 64
  5.2 Comparing the overhead of profile collection ............. 66
  5.3 Accuracy comparison ......................................... 70
    5.3.1 Reducing the sample rate ............................... 73
  5.4 Benefits of extra information ................................ 75
  5.5 Summary ...................................................... 78

6 Conclusions and future work 79
  6.1 Profiler Design Analysis ..................................... 79
  6.2 Applications .................................................. 80
  6.3 Future work ................................................... 81
    6.3.1 Extensions ................................................. 82
    6.3.2 Implementation improvements ............................ 84

Bibliography 87
Tables

1.1 Results from running the small example with gprof ............... 4

3.1 Trampoline insertion summary ........................................ 33

5.1 Compilation options for our experiments. .......................... 65
5.2 SPEC benchmark results .............................................. 67
5.3 Distortion and dilation for csprof and gprof on the SPEC benchmarks 72
5.4 Distortion caused by csprof and gprof on the SPEC benchmarks .. 76
Chapter 1

Introduction

Computer programs are becoming increasingly important in science and engineering, used for such diverse tasks as sequencing the human genome, performing climate simulations, and validating engineering designs. The speed of these programs is important to their users, as the speed is directly proportional to the amount of work they can accomplish. A fast program implies that more and/or larger simulations can be examined or more designs can be tested before deciding on a particular design. Conversely, a slow program means that results for simulations and testing may take weeks or months to be delivered instead of days. The desire for and importance of fast programs in these disciplines motivates the need for performance tuning.

Identifying areas of resource consumption, or profiling, is an old discipline. People have always wanted their programs to run faster. An early study in profiling showed that the majority of a program’s time is concentrated in a very small amount of code [23]. However, even an expert programmer’s predictions as to which portions of the code are “hot” are often very poor. Profiling, then, is usually done via monitoring code that runs concurrently with the application under study to determine which areas of the code are “hot.”

Designing a good profiler is a matter of determining on what the programmer wishes to focus and then constructing a profiler to deliver that information to the programmer. Several profilers [3, 36] focus on indicating “hot functions” to the programmer as an indication of places to look for bottlenecks; this type of profiler is known as a flat profiler. While this information is helpful, merely knowing which functions are “hot” is often not enough. If a function’s execution time depends on
the size of its arguments and the function is called from several different places in the program, then truly understanding the performance of the function implies understanding the calling context of the function. Furthermore, knowing the immediate callers of a function is not necessarily enough for effective performance analysis. Multiple levels of libraries may be used by one application, and both the libraries and the application may each have several levels of abstractions. Understanding why a low-level library routine, e.g. `memcpy`, the standard library routine for copying sequences of bytes, is consuming a majority of a program’s execution time requires knowledge of the full calling contexts of that system routine.

The above scenario and others like it point to the need for a different kind of profiler: one that collects calling context information. Such a profiler is known as a call path profiler and is the topic of this thesis. Previous call path profilers have collected calling context information by adding instrumentation to the program being profiled: extra code at function entry and/or exit to record the dynamic behavior of the program. Rather than collecting full call path information, however, the standard call graph profiler design, `gprof` [19], augments each function in the program with instrumentation to count calls to that function from its caller(s). In this manner, only one level of calling information from each function is collected, hence the name “call graph profiler.” `gprof` also periodically samples the program counter to obtain a flat profile of the program. These two pieces of information are combined to estimate the time spent in each function call from a particular function. This combining is done via the average time assumption, which assumes that all calls to a particular function are equally costly. We will illustrate that this assumption is a poor assumption with a short example program, shown in Figure 1.1.

Figure 1.1 is a trivial program, but it highlights two attributes which modern, modular codes exhibit. First, the program executes a large number of function calls. Object-oriented, pattern-based software development methods are becoming increasingly important and such methods tend to produce programs that make a large num-
#define HUGE (1<<28)
void d() {}
void c(int n) { int i; for(i=0; i<HUGE/n; ++i) d(); }
void b(void (*f)(int)) { f(2); f(2); f(2); f(2); }
void a(void (*f)(int)) { f(1); f(1); }
int main() { a(c); b(c); return 0; }

Figure 1.1: A short program that demonstrates the shortcomings of gprof's average time assumption.

The number of function calls. Instrumenting every call to collect profile data for such programs incurs prohibitive overhead. Second, the program exhibits context-dependent behavior: the time c takes to execute is dependent upon its first argument. Thus calls to c from b (through the function pointer f) take a different amount of time than calls to c from a (through the function pointer f). Abstractions exposed by libraries, such as hash tables, are designed to be used for multiple purposes in different parts of the program; a good call path or call graph profiler needs to capture the full calling context of calls to such abstractions to provide the performance analyst with the necessary information to evaluate which parts of the program on which to focus tuning effort.

We profiled this small program via gprof to illustrate some of the problems with its approach. By simple calculation, a call to a and a call to b should take the same amount of time. However, gprof attributes time spent in c to its callers a and b by the number of calls to c. As there are twice as many calls to c from b as from a, gprof attributes two-thirds of the time spent in c to b and one-third to a. gprof does not provide accurate information for context-dependent calls.

Furthermore, obtaining this poor information is expensive. Table 1.1 shows the dilation factors observed by using gprof-style instrumentation on a variety of platforms and compilers. While Figure 1.1 is an extreme case, we show in Chapter 5 that dilation factors similar to those shown in the table are not only observed on toy programs,
<table>
<thead>
<tr>
<th>Platform</th>
<th>Dilation factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacOS X/PowerPC 7450 1.33 GHz</td>
<td>4</td>
</tr>
<tr>
<td>Linux/Pentium4 2.0 GHz</td>
<td>3</td>
</tr>
<tr>
<td>Tru64/Alpha 667 MHz</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 1.1: Results from running Figure 1.1 on various platforms with compiler-based \texttt{gprof} instrumentation added. A dilation factor of 2 means an instrumented binary took twice as long to run as an uninstrumented binary.

but “real-world” programs as well. Note also that adding more instrumentation in an attempt to record more accurate information (e.g. full call path information) would only solve half of \texttt{gprof}’s problems on this small code. Our small example does not depend on external libraries; if it did, however, the instrumentation added by \texttt{gprof} would be unable to monitor calls made into and inside of the libraries.\footnote{Unless pre-instrumented versions of the libraries are available.} The combination of high overhead, inaccurate profiles, and inability to profile external libraries makes \texttt{gprof} and derivative techniques problematic for profiling large-scale, modular codes—precisely the codes which need call path profiling for effective tuning.

This thesis describes the design of a call path profiler that addresses the above problems in the following ways. First, the profiler we describe requires no extra support from the compiler; the information generated by standard system compilers for debugging suffices and no recompilation is required. Second, no instrumentation is inserted into the binary either before or during its execution. The profile is collected through stack sampling, which enables the user to tune the overhead and precision to the requirements of the application. This strategy avoids high monitoring overhead for call-intensive programs and avoids severely inflating the time attributed to small, frequently-called functions. Third, data about the application along with its associated libraries (including those dynamically loaded at runtime) is collected au-
tomatically. Fourth, by transparently modifying the execution state of the program, we are able to record the return of each frame present when a sample occurs. This enables us to precisely attribute samples to call paths. As an added benefit, this strategy enables us to avoid re-examining frames which have been seen during previous samples and thus to reduce the overhead of call path collection. Taken together, I contend that these features can be used to construct a call path profiler that will provide superior accuracy and low overhead without modifying the profiled program.
Chapter 2

Related work

The need for collecting profiles that attribute performance metrics to functions while simultaneously attributing the costs to the calling context of those entities has long been recognized. Such a profile is known as a call path profile. In a call path profile, data about individual functions is collected, as in the flat profile, but information about the context of each function call in the program is also collected. "Context" can range from the immediate caller of the function in question to its entire active call chain. In this chapter, work related to three major areas in this field is reviewed: data structures for such profilers, profilers that collect information about calling context via instrumentation, and profilers that collect the needed information via sampling. Along the way, the work of this thesis is noted, compared, and contrasted with previous work when appropriate.

2.1 Data structures

To represent calling context in a context-sensitive profile, one useful data structure is the dynamic call tree (DCT), a rooted tree in which every node represents a unique procedure activation and edges between nodes represent function calls. Note that even if a function is called multiple times from the same context, a new node is created at each invocation to represent the function. Therefore, while a DCT provides a complete picture of the dynamic behavior of the program, its space costs make it prohibitive to collect. Two approximations to the DCT are commonly used in practice: the dynamic call graph (DCG) and the calling context tree (CCT).

To construct the DCG from the DCT, create a node in the DCG for procedure
If a node for procedure $p$ exists in the DCT. Then add edges between the nodes according to the following rule: if there exists an edge between procedures $p$ and $q$ in the DCT, add an edge between the node representing $p$ and the node representing $q$ in the DCG. While the DCG is much more compact than the DCT, resources consumed by a procedure $p$ are not distinguished by the context in which $p$ is called.\footnote{Ammons, Ball, and Larus\cite{1} call this the “gprof problem”; we will see why this is so below.}

As a compromise between the excessive space usage of the DCT and the inability of the DCG to attribute appropriate profile data to different calling contexts, Ammons, Ball, and Larus devised the CCT.\cite{1} The CCT collapses equivalent nodes in the DCT: two nodes are equivalent if they each represent the same procedure and they each have the same parent node. This aggregation discards redundant context information, as the DCG does, but it preserves the unique context of the calls provided by the DCT. Figure 2.1 shows how the three data structures differ. While the breadth of the CCT is bounded by the number of procedures in the program, the depth of the CCT is not, due to recursive procedures. Changing the test for equivalence for nodes in the DCT can provide an upper bound on the depth of the CCT at the cost of discarding some accuracy in the collected profile information. The profiler in this thesis collects the full CCT, and the potential of deep trees has not been a problem in practice.

## 2.2 Instrumentation-based profilers

One method of collecting call path profiles is by adding instrumentation to some or all of the functions in the target program. The purpose of this added code is to record performance metric(s) of interest, such as time spent in functions, cache misses incurred by functions, and so forth. Instrumentation-based profiling has the advantage of being complete, in that a precise picture of the program’s execution can be obtained. In addition, added instrumentation has great flexibility in what is recorded. Metrics such as the number of function calls are easily obtainable; metrics
such as execution time or those provided by hardware performance counters can be accumulated on a function level or a basic block level.

The first call graph profiler was \textit{gprof}, developed by Graham, Kessler, and McKusick [19]. To profile a program using \textit{gprof}, a user instructs the compiler to instrument each function being compiled with code at the function’s entry. This instrumentation records information about the caller of the function (upon function entry). Call counts can then be assigned to the edges in the dynamic call graph. \textit{gprof} augments this graph with a flat profile taken while executing the program, apportioning time to each function in the call graph based on the number of calls to it and the flat profile.

While \textit{gprof} was and is useful, its design suffers from two flaws which render it unsuitable for profiling large, modern applications: its reliance on the compiler to
insert instrumentation code and its assumption about how time is distributed among calls. *gprof*’s reliance on the compiler to insert instrumentation means that if a user wishes to profile an application, that application must be recompiled for the compiler to insert the necessary instrumentation. On large programs, compile times of several hours are not uncommon, making recompilation to profile a heavy burden to bear. Modern applications also depend heavily on *dynamic libraries*: collections of shared code used by several applications on the system. *gprof* will not insert instrumentation into these libraries, and recompiling all the libraries used by an application to obtain profile data is either impractical, adding to the already large recompilation time of the application, or infeasible, since system libraries generally do not come with source code.

Research in binary instrumentation-based profilers can be seen as addressing the problems in *gprof*’s design. While compilers still offer flags to insert instrumentation during compilation, tools such as the Executable Editing Library (EEL) [24], Etch [30], and ATOM [32] are general-purpose tools for instrumenting a program binary at arbitrary points. This instrumentation is done after compilation and can therefore be done without source code; inserting the code necessary for a *gprof*-style profiling run is a simple matter. In addition, with such tools, instrumentation can also be added to the libraries used by a program, enabling collection of more informative profiles.

Another point at which instrumentation can be inserted is during the program’s execution. Interfaces such as DynInst by Buck and Hollingsworth [8] or systems built on DynInst, like the Dynamic Class Probe Library (DCPL) by deRose, Hoover, and Hollingsworth [13], enable dynamically constructed snippets of code to be inserted at arbitrary points during execution. This *dynamic instrumentation* usually requires the target program to be run under the control of another application due to restrictions on program self-modification imposed by modern operating systems.
Paradyn by Miller et al, designed for profiling large-scale parallel programs, is a flexible tool that is based on dynamic instrumentation [27]. Paradyn seeks to discover answers to three primary questions: why is the program running slowly, where is the slowdown incurred, and when does the slowdown happen? Instrumentation is inserted or removed at run time to answer these questions. Paradyn searches for performance bottlenecks according to a static ordering of modules and functions within modules; Cain, Miller, and Wylie describe an alternate strategy in which the search is driven from an application’s callgraph instead [9]. While Paradyn automates the search for performance problems, there is no reason why the user could not personally conduct the search, directing the tool to instrument or de-instrument portions of the program [7]. Another possibility that dynamic instrumentation admits is to use an inexpensive technique, such as program counter sampling, to determine hotspots. Instrumentation can then be added around the hotspot to more precisely identify the bottleneck [15].

*gprof’s second flaw lies in its assumption made about distributing time among callers. *gprof assumes that the time spent in each invocation of a procedure in the program is independent of the context in which the procedure is called; this assumption is known as the average time assumption. If this assumption is false, however, *gprof will report inaccurate results [33, 29]. Programs with many small functions, higher-order functions, or recursive procedures are particularly prone to violating the average time assumption.

Other researchers have sought ways to provide more accurate profilers while following the basics of *gprof’s approach. Ammons, Ball, and Larus [1] explain how to construct a CCT by instrumenting a program. To control the depth of the collected CCT, backedges are introduced into the CCT. When a function $f$ calls a function $g$ and $g$ exists as an ancestor of $f$, then an edge from $f$ to the ancestor occurrence of $g$ is created, rather than creating a child node of $f$. These backedges make the CCT
decidedly un-tree-like and toss away some of the context provided by the original formulation of the CCT.

Spivey [31] follows the same approach as gprof, but instead of the instrumented function recording its caller only, the added instrumentation keeps track of the set of active functions at the point of the call. This set is then used to dynamically construct a finite state machine which handles accurate attribution of resource consumption. Initial results indicate that Spivey’s scheme performs similarly to gprof, despite collecting more data; Spivey notes that his implementation has not been extensively tuned for performance. Conway and Somogyi’s work with deep profiling in the Mercury compiler [10] uses similar techniques combined with compiler analysis to reduce the cost of profiling and preserve expected behavior while profiling (e.g. tail calls). Both approaches give more accurate information that gprof, but do little to address the high overhead of instrumentation.

CATCH [14] combines a post-compilation instrumentation phase with an efficient method of tracking the current state of the application. During the instrumentation phase, CATCH builds a static call graph of the application; tracking the current state of the application is then done by maintaining a node corresponding to the currently active procedure. This node is then updated at procedure call and return. To deal with recursive functions, CATCH also maintains a shadow stack of nodes in one-to-one correspondence with the active call stack. In addition, CATCH also provides for profiling only certain parts of the call tree, through selection of interesting subtrees or specifying that certain subtrees should not be profiled. While CATCH provides for accurate and precise information about the dynamic behavior of the program, the static call graph built by CATCH does not detect indirect call sites. This limitation will not be noticed for most programs, but it does make CATCH unsuitable for programs written in C++ and similar languages.

A third problem, not addressed by any of the above work, but which is a fundamental issue with instrumentation-based profilers, is that of the probe effect. The added
instrumentation necessarily increases the time taken to execute an instrumented procedure. For small and/or frequently-called procedures, this added instrumentation can excessively distort the time attributed to such procedures. Our results in Chapter 5 show severe distortion for call-intensive codes with small functions.

2.3 Sampling-based profilers

While profilers that rely on instrumentation can provide precise information for all parts of an application, the cost of executing the added instrumentation can be unacceptably high. A doubling of execution time is common [15, 31] and experiments with compiler-inserted instrumentation described in Chapter 5 support those conclusions. Furthermore, the added instrumentation always executes, making the overhead fixed and unadjustable.

One idea for reducing the cost of instrumentation was proposed by Arnold and Ryder [4]. Instead of instrumenting the entire application, Arnold and Ryder add checking code at particular points. The checking code is so named because it checks to see if some condition is met at that point in the program; if so, the checking code switches to an instrumented version of the code, which then runs and eventually switches back to the uninstrumented version of the code. Backedges in the control flow graph of the program always switch to the uninstrumented version of the code, bounding the amount of time the instrumented code runs. Overhead can then be controlled by changing the condition in the checking code. They show that an accurate profile can still be collected with this technique, even though the instrumented code is not always active.

This idea of “checking code” or a “trigger” is an application of an alternative method to instrumenting programs: statistical sampling. Statistical sampling periodically interrupts the program to inspect the current state of the program and to record profile information. If enough samples are taken, then an accurate picture of the program’s execution will emerge. While precision suffers, sampling has the signif-
icant benefit of being much less intrusive than instrumenting the program, although the flexibility of sampling is somewhat less. For call path profilers, taking a sample typically involves the recording instruction pointers in the current call chain, which are gathered by unwinding the call stack. To provide for the greatest flexibility in profiling, some systems will provide for both statistical and instrumentation-based profiling: Liang and Viswanathan [25] describe such a profiling interface for the Java Virtual Machine.

Several call path sampling-based profilers have been described in the literature. Hall and Goldberg [21] describe a framework very similar to that described in this thesis. They observe that any resource can be used, subject to the constraint that the resource is monotonic—that is, the use of the resource over the entire program is equal to the sum of its usage by the procedures in the program. This observation then drives the idea of a “generalized interval timer” which can monitor one or more system resources, such as page faults, bytes read/written, wall clock time, etc., and takes a sample every “gtick”. A complete stack sample is taken at each gtick and the collected stack samples are aggregated for offline analysis. The profiler design in this thesis accommodates any monotonic sample source.

OProfile [17] is a system-wide sampling profiler for machines running the Linux kernel. It records samples in user code as well as kernel code. In this respect, it is quite similar to DCPI [3], which implements similar functionality for Alpha-processor based machines. If running on a Linux 2.6 kernel, OProfile can also collect call path profiles; call paths can be collected across the user/kernel boundary, just like regular samples. However, the call path profile collection depends on the profiled code have been compiled with frame pointers (in addition to the normal stack pointer). If the profiled code does not use frame pointers, then OProfile will only collect flat samples.

Whaley created a sampling-based profiler for the Java Virtual Machine with an eye towards profitably using its information to support dynamic method compilation [35]. Like Hall and Goldberg, Whaley uses stack unwinding. Two key features in Whaley’s
profiler stand out. First, he limits the space required by a CCT in the presence of recursion by unwinding only a fixed number of stack frames at each sample; his data structure, the *partial calling context tree* (PCCT) is necessarily incomplete because of this. Second, he marks each frame visited by the unwinder with a *sample bit*; unwinding could then halt early if a visited frame was found. In addition to halting unwinding early, the sampled bit also enabled efficient insertion of stack samples in the PCCT. Each recorded frame in the stack sample remembered its node in the PCCT; when a visited frame was found, insertion of the collected stack sample could begin at the node associated with the frame in question, rather than beginning at the root. This combination of features delivered low overhead (2%-6%).

Arnold and Sweeney also followed the stack sampling route to collect an approximate CCT (ACCT) [5], but sought to reduce the cost of stack sampling. Their Java implementation instruments all method entries with to check a global method call counter; a stack sample is taken when the counter exceeds a certain threshold (this is a twist on the idea of checking code presented earlier). During the sampling process, the return address of each visited stack frame is replaced with the address of a known function: a *trampoline*. This function is so named because control is briefly transferred to the trampoline before returning to the real caller. This idea is similar to Whaley’s sample bit and permits the same optimizations in stack unwinding and sample insertion to be made. The replaced return addresses are retained in profiler-private storage. When a method whose frame has been sampled returns, the trampoline locates the correct return address *ra* and returns to *ra*. No results were given for the overhead of their method; given how similar their method is to Whaley’s, similar overhead is to be expected.

This thesis presents a profiler which also uses stack sampling. Whaley’s sample bit (and Arnold and Sweeney’s trampoline refinement of the same) is adopted to reduce the cost of stack unwinding. Unlike Whaley’s profiler, however, our profiler samples the entire call stack, rather than a fixed-size portion of the stack. Unlike Arnold and
Sweeney trampoline insertion strategy, our profiler has at most one active trampoline at all times; this change simplifies the stack unwinding process. Neither Whaley or Arnold and Sweeney provided a detailed explanation of how their sample bit was to be inserted for “real world” programs; this thesis gives a complete description of inserting the trampoline, as well as strategies for dealing with highly optimized code and non-local exits. Such a description can be found in the next chapter.
Chapter 3

High-level profiler design

This chapter describes our high-level, portable design of a call path profiler that collects its data by call stack sampling. Call stack sampling provides controllable overhead and captures the full calling context of each sample event. We discuss methods for addressing three key problems of collecting calling context. First, we describe a general framework for collecting samples. Second, we present a technique to improve the efficiency of stack unwinding by using a \textit{sentinel}; this sentinel enables us to halt stack unwinding early in many cases. We conclude the chapter with a collection of strategies for dealing with advanced program features and optimizing compilers.

3.1 How to collect samples

The design presented in this section collects \textit{call stack samples}. A call stack sample is a list of instruction pointers representing the active procedures at an arbitrary point in the program.

Before describing the mechanisms involved in collecting call stack samples, we define a few terms. A \textit{procedure activation record} or simply an \textit{activation record} denotes the space where a procedure stores its local variables and its return address. We will also have cause to refer to activation records as \textit{call frames} or simply \textit{frames}. A \textit{register frame procedure} maintains its return address in a machine register; it may or may not allocate space on the stack for local variables. A \textit{stack frame procedure} always allocates at least some space on the stack and stores its return address on the stack. A procedure \textit{prologue} refers to a linear sequence of instructions at the beginning
of a procedure; the end of a prologue is determined when the return address has been moved to its saved location (whether that location is a register or a memory location on the stack). A procedure epilogue consists of a linear sequence of instructions, the first of which moves the return address out of its saved location in preparation for returning from the procedure. The last instruction of a procedure epilogue exits the procedure. There may exist multiple epilogues in a given procedure. Finally, the body of a procedure consists of all those instructions that are neither in the prologue or epilogue.

3.1.1 Sample events

Call stack samples are gathered in response to sample events, events that signal the consumption of a resource of interest to the profiler. There are two categories of sample events: synchronous and asynchronous. Synchronous events are events that occur as the direct result of program action, e.g. allocating memory or directing data to be sent over the network. Synchronous events can be monitored by having a surrogate routine that counts some metric of interest (e.g. bytes allocated), calls the sampling routines when appropriate, and forwards the call to the actual library routine. Taking a sample requires the context of the calling routine: a structure containing the values of machine registers at the time of the event. A context can easily be obtained by calling library routines, such as getcontext.

Figure 3.1 shows a malloc replacement that records the current call path every 10000 bytes allocated. The profiler state maintains a count of the number of bytes allocated during a thread’s execution. This count is this incremented and checked to see if it exceeds the sample period. If so, malloc’s caller’s context is determined, the number of samples which should be attributed to this context is determined, and the call path leading to this context is computed. The system’s malloc routine is then called, presumably through a function pointer (section 3.3.5), to actually allocate the memory. The entire process is transparent to the application.
void *
malloc(size_t nbytes)
{
    prof_state_t *state = get_profiling_state();
    state->bytes_allocated += nbytes;

    /* check to see if we should take a sample */
    if(state->bytes_allocated >= 10000) {
        context ctx;
        unsigned long count;

        getcontext(&ctx);
        virtual_unwind(&ctx);

        count = state->bytes_allocated / 10000;
        state->bytes_allocated = state->bytes_allocated % 10000;

        record_sample(state, &ctx, count);
    }

    return system_malloc(nbytes);
}

Figure 3.1: A replacement for malloc that takes a call stack sample every 10000 bytes allocated.
void
signal_handler(int signal, siginfo_t *info, void *context)
{
    prof_state_t *state = get_profiling_state();

    record_sample(state, context, 1);
}

Figure 3.2 : A possible POSIX signal handler for an asynchronous event.

The design presented in this thesis also supports asynchronous events. Asynchronous events are events that are not triggered by direct program action. Examples include interval timers under Unix or the overflow of hardware performance counters; note that the overflow of hardware performance counters may or may not be exposed to the program. A program is generally unaware of asynchronous events unless it specifically requests that it be notified of them. Notification of asynchronous events usually takes place via a mechanism such as POSIX signals. Regardless of the specific notification mechanism, the operating system should be capable of providing a context for the event. The instruction at the program counter is then understood to "consume" the metric.

Figure 3.2 shows an example of a POSIX signal handler for an asynchronous event. Profiler state is retrieved via get_profiling_state, as there may be multiple states, e.g. in a multi-threaded application (see section 3.3.7). As with the synchronous sample event handler, record_sample is called to determine the call path to the current context.

3.1.2 Taking a sample

Collecting call stack samples is done by unwinding the stack: starting from an initial context, the current stack frame is examined to determine the previous stack frame. Figure 3.3 shows pseudo-code for collecting a call stack sample. The instruction
list_t *
collect_stack_trace(context_t *context)
{
    /* make a copy, since 'context' may be coming from a
    signal handler */
    context_t copy;
    list_t *trace = create_list();
    void *pc;

    memcpy(&copy, context, sizeof(context_t));
    pc = get_register(&copy, REG_PC);

    do {
        push(pc, sample);
        unwind(&copy);
        pc = get_register(&copy, REG_PC);
    } until(pc == 0);

    return trace;
}

Figure 3.3: Simple algorithm for collecting a call stack sample.

A pointer of the current context is recorded and the examination proceeds to the calling
procedure’s context, which then becomes current. This process is repeated until the
context prior to the main procedure of the program is identified; such a context is
usually identified by having an instruction pointer of zero.

The list of instruction pointers identified by unwinding is then stored, to be written
out for later analysis after program termination. Two approaches to storage are
possible. The first approach, used by the implementation described in Chapter 4, is
to store the collected samples in a calling context tree (CCT) [1]. Doing this captures
the data sharing that two consecutive samples are likely to exhibit. This approach,
however, throws away temporal information; once samples s₁ and s₂ are inserted into
the CCT, it is impossible to tell the order in which they occurred. Knowing the order
in which samples are taken may be useful in some contexts: if a function using a
cache shows up as a bottleneck, partitioning its profile into "cache-cold" and "cache-
hot" samples would be helpful. To capture this temporal information, each sample
may be stored in its entirety; instead of a CCT, then, the profiler simply stores a list
of samples. Periodic flushing of the sample buffer to disk may be necessary as the
second approach requires much more memory than the CCT approach. The second
approach was used by Hall [21]. Regardless of which is used, storage for the collected
call stack samples resides in the profiler state for each thread of control.

If samples are stored in a CCT, then an efficient implementation of storing child
nodes at each node is required, since stack sampling collects instruction pointers
rather than function identifiers. Consider a function $f$ which calls a long-running
function $g$. For each sample taken in $g$, the children of the node associated with $g$’s
callsite in $f$ will need to be searched to determine whether the instruction pointer
at the end of the stack sample has been sampled before. As there many be many
samples taken in $g$, it is critical that this determination be as efficient as possible.
To strike a balance between lookup efficiency and space efficiency$^1$, we recommend
maintaining the children of a node as a balanced tree.

\section{Efficient unwinding with a trampoline}

In this section, we present a method to increase the efficiency of recording call stack
samples. While the algorithm in the previous section is simple and works well, per-
forming a full unwind of the call stack for each call stack sample is inefficient. All
callers above the least common ancestor with the previous sample are already known;

$^1$Using asynchronous events on a POSIX system implies that \texttt{malloc} cannot be used to allocate
memory. To safely allocate memory from a signal handler, a profiler must manage its own memory
using \texttt{mmap}. To avoid wasting memory as we collect data, we prefer to use data structures which do
not require discarding old data as the amount of data stored grows (e.g. hash tables).
repeatedly unwinding these activation records is inefficient. Consider a program where
the majority of its execution is spent in one routine that is called only a handful of
times. At each sample event, time would be wasted by unwinding the call stack to
the main routine when little has changed on the call stack.

To address this inefficiency, our profiler borrows the "sample bit" idea of Wha-
ley [35]. The idea is to place a bit of information in each activation record which
indicates whether the associated procedure has been sampled at some point in the
past. New activations records are created with this bit clear. To collect a call stack
sample, then, a profiler only needs to unwind activation records, marking as it goes,
until a procedure activation with its sample bit set is found. At this point, the pro-
filer can be certain the least common ancestor between the current call stack and the
previous call stack has been located. The common prefix between the two samples is
then concatenated with the new suffix and the complete call stack sample is recorded.

Assuming that the hardware masks out the sample bit when the data in the activa-
tion record is accessed, the approach works very well. Given the previous assumption,
on word-addressed systems, the sample bit is placed in the low-order bits of the return
address. The hardware then ignores those low-order bits when returning from a func-
tion. If the hardware does not ignore the sample bit, the program under examination
can be modified to perform the necessary masking itself. For each function, all of the
return instructions are located. Prior to each return instruction, an instruction which
masks off the sample bit can be inserted. While this involves modifying the program
binary (and all of its component libraries), this workaround retains all of the essential
features of the sample bit approach but with very low overhead.

Arnold and Sweeney [5] retained the desirable features of the sample bit approach,
but discussed an alternate implementation that works in environments where the
hardware does not perform the necessary masking, and modification of the program
binary is undesirable. Instead of flipping a bit in the return address, Arnold and
Sweeney replace the return address of all unwound frames with the address of a
known function or a trampoline. The overwritten return addresses are saved for later use. When a procedure whose activation record has been modified by call stack sampling returns, the trampoline is activated. The trampoline then looks up the most recently replaced return address and jumps to that address. Using a trampoline as the sample bit incurs overhead when a sampled procedure returns, but it does not require modifying the program binary.

Our design differs from Arnold and Sweeney by maintaining only one trampoline at the top of the sampled call stack, moving it on procedure returns and moving in when a sample event is being processed. Figure 3.4 shows the movement of the trampoline during the execution of a program.

Using a trampoline also permits one further optimization to call stack sampling. Once the least common ancestor between the current call stack under examination
and the previous call stack sample has been located, the complete call stack sample must be recorded. But only a suffix of the actual call stack has been unwound. A search must then be made in the recording structure to determine the prefix to concatenate with the collected suffix. When using a sample bit, this search must be performed on every sample taken. When using a trampoline, however, a profiler can maintain the needed prefix (a.k.a. “shadow stack”) automatically; this cached call stack is updated at every sample and the trampoline pops one node off the shadow stack each time it is activated.

3.2.1 Marking an activation record with a trampoline

Whichever approach is chosen for increasing the efficiency of stack unwinding, locating the return address is of paramount importance. In this section, we consider a systematic way to locate the return address for any procedure. We assume that each procedure in the program dedicates a register to hold the frame pointer and a register to hold the stack pointer. These registers may be identical if the frame pointer and the stack pointer for a procedure are identical; this situation occurs if a function allocates a fixed amount of stack space. Two events trigger a trampoline insertion: when a sample event occurs and when a procedure whose activation record has been instrumented with the trampoline returns.

Procedure descriptors

We also need a mechanism to retrieve information about the compiled procedures in the program; this information will come packed in small structures known as procedure descriptors. While procedure descriptors are most often compiler-generated and used for exception handling, the profiler design presented in this thesis is not dependent upon compiler-generated procedure descriptors. Procedure descriptors could be synthesized by examining the program binary (as well as any libraries) at load time or on the fly as the program runs. We assume the existence of a func-
tion, `get-procedure-descriptor(addr)`, which returns a procedure descriptor corresponding to the address `addr`.

To support trampoline insertion, procedure descriptors must support a minimal interface, described below. Certain operations may not be applicable to certain procedure descriptors. For instance, the operation `save-register`, indicating the register that holds the saved return address during the procedure’s body, does not make sense for procedures that save their return address on the stack. Operations that make sense only on particular kinds of procedure descriptors will therefore be noted as necessary. In the discussion which follows below, `pd` represents the procedure descriptor of an arbitrary function `f`. The interface we describe borrows heavily from the `libexec` interface present on Irix and Tru64 systems.

- `stack_frame.p(pd)`: return true if `f` is a stack frame procedure;

- `ra_save_offset(pd)`: returns the offset from the frame pointer at which the return address is saved. This function allows the profiler to determine the location in the stack frame where the return address is located. This operation only applies to stack frame procedures;

- `frame_size(pd)`: returns the frame size of `f`. This operation should return 0 for a register frame procedure’s descriptor. This information will be used if `f`’s frame has not yet been allocated to compute the location where the return address will be saved in `f`’s frame;

- `ra_slot_offset(pd)`: returns the offset from the base of `f`’s frame to the location where the return address is saved;

- `prologue_length(pd)`: returns the length of `f`’s prologue in bytes. Knowing the length of the prologue enables the profiler to determine whether the instruction pointer at which a sample event occurs lies within `f`’s prologue code or body/epilogue code;
- **entry_register**(pd): return the register that contains the return address upon entrance to \( f \). This register is usually prescribed by the calling conventions, but the compiler may determine that a more efficient (non-standard) calling sequence can be obtained by changing the register.\(^2\) This piece of information enables the profiler to find the return address if it has not yet been saved into \( f \)'s frame. In parts of the discussion below, it will be convenient to refer to this register as $ra$;

- **save_register**(pd): for a register frame procedure, return the register in which the return address is saved during the body of the procedure. This register may be the same as the result of **entry_register**(pd). If \( f \) is a register frame procedure, **save_register** indicates where the return address may be found during execution of \( f \)'s body;

- **stack_set_offset**(pd): return the offset in bytes from the entry point of \( f \) to the instruction that allocates the stack frame for \( f \). This operation is not supported for register frame procedure descriptors. With this offset in hand, determining whether \( f \)'s stack frame has been allocated is a simple matter;

- **store_return_address_offset**(pd): return the offset in bytes from the entry point of \( f \) to the instruction that stores the return address into \( f \)'s frame. Similar to **stack_set_offset**, this bit of information will be used by the profiler to check whether or not the return address has been saved to \( f \)'s frame;

- **address_in_epilogue_p**(pd, addr): return true if the address \( addr \) lies within an epilogue of \( f \). This function enables the profiler to determine when a procedure is executing its epilogue. As the compiler may freely insert function epilogues as it deems necessary, arbitrary points in the procedure may be part of a function epilogue.

\(^2\)In practice, an alternate register for the return address is usually used when one register frame procedure calls another.
Several of the above operations deal with offsets from the start of the function $f$. Doing so requires that the start address of $f$ be readily available. Therefore, a function `function_start_address(addr)` that returns the address of the entry for the function containing $addr$ is necessary.

As mentioned above, procedure descriptors can be synthesized by examining the instruction stream of the program. Therefore, if some of these operations are not available, the instruction stream of the program can be analyzed to provide appropriate values; the values thus obtained can be cached for later use.

**Performing the insertion of the trampoline**

The process of insertion is divided into two parts: locating the place where the return address is stored and then replacing that location with the address of the trampoline (or adding the sample bit to the return address). Procedure descriptors are used to assist in determining where the return address is located for a given procedure at a given point. Since the return address may reside in a machine register, the contents of the machine registers must be available for inspection and modification. Since sample events can occur during a procedure’s prologue—when the return address has not been moved to its saved location—two locations are computed during the first part of trampoline insertion. The first location is where the return address is currently located; the second location indicates where the return address will reside during the body of the procedure. Identifying the second location is necessary to “undo” the trampoline insertion at the next sample event. These two locations may coincide if the sample event occurs during the body or epilogue of a procedure.

Figure 3.5 shows the definition of two structures used during the first phase of trampoline insertion.

Once a struct `locinfo` is filled in, inserting the trampoline reduces to a straightforward case analysis on its `current` member. The information from `stored` is transferred to the profiling state for use at the next sample event.
struct raloc {
    enum { REGISTER, ADDRESS } type;
    union {
        int register;
        void **address;
    } location;
};

struct locinfo {
    struct raloc current, stored;
};

Figure 3.5: C structure definitions for structures used to store information about the location of the return address for the current activation record.

Inserting the trampoline for register frame procedures will be discussed first, because register frame procedures are less complicated than stack frame procedures. We need to consider two classes of register frame procedures: those that save the return address in a different register than the one that the return address occupies at procedure entry and those that leave the return address in the register it occupied at procedure entry. In the latter case, the location of the return address is constant throughout the execution of the function and can be easily identified from platform ABI conventions. Nothing more needs to be done for this case.

Figure 3.6 contains pseudo-code for determining where the return address is in a register frame procedure. find ra in regframe is straightforward: the procedure descriptor for the given context is retrieved and then used by the predicates in prologue and in epilogue to determine whether the procedure is in its prologue, epilogue, or body. Pseudo-code for in prologue is given in Figure 3.7. in epilogue is a simple application of address in epilogue.p, described earlier.

Locating the return address location for stack frame procedures is more complicated, but follows the same outline as for register frame procedures: a case analysis based on whether a procedure is executing its prologue, body, or epilogue. Figure
void
find_ra_in_regframe(prof_state_t *state, struct locinfo *loc,
    void *context)
{
    void *pc = get_register(context, REG_PC);
    proc_descr_t *pd = get_procedure_descriptor(pc);

    loc->current.type = loc->stored.type = REGISTER;
    loc->stored.location.register = save_register(pd);

    if(in_prologue(pd, pc) || in_epilogue(pd, pc)) {
        loc->current.location.register = entry_register(pd);
    } else {
        /* we are in the body of the procedure */
        loc->current.location.register = save_register(pd);
    }
}

Figure 3.6: Pseudo-code demonstrating how to determine the current return address location for a register frame procedure.

int
in_prologue(proc_descr_t *pd, void *address)
{
    /* figure out where the function starts */
    void *start_address = function_start_address(address);
    void *body_begin = start_address + prologue_length(pd);

    return (start_address <= address) && (address < body_begin);
}

Figure 3.7: Pseudo-code for determining whether an address lies within a function's prologue.
3.8 shows pseudo-code for locating the return address in a stack frame procedure; Figure 3.9 shows a helper procedure which handles the case when the procedure is executing its prologue. The complications stem from the multiple states that must be tested during the prologue. We must perform different actions based on whether the stack frame has been allocated and whether the return address has been stored into the stack frame. Handling the body and epilogue cases are as straightforward as the register frame procedure case.

In summary, then, Table 3.1 describes the cases and actions taken by the profiler depending upon what information is discovered by examining the procedure’s procedure descriptor. When using an asynchronous sample source, it is possible to have “timing windows,” where the recorded location will be incorrect if the next event occurs very shortly after one has just been processed. For example, if a function is interrupted during its prologue before allocating its stack frame, the profiler will record its address as being on the stack. If the next event occurs before the function actually allocates its stack, the profiler will attempt to remove the trampoline from the stack and will fail. This timing window is generally very small (a handful of instructions) and is unlikely to be a problem on modern gigahertz processors. However, in the event that it does happen, the techniques described in section 3.3.3 will handle it adequately.

The above discussion addressed the fully general case of sample bit/trampoline insertion. If a profiler is using the trampoline approach, then the trampoline must also be inserting when returning from a marked procedure activation. When such a return is made, the procedure activation being returned to is always executing its body. This knowledge enables streamlined code to be written for this special case.

**Call-path edge counting**

Recording call stack samples is useful for identifying the context of performance problems, but knowing the context alone does not provide sufficient information to ac-
void
find_ra_in_stackframe(prof_state_t *state, struct locinfo *loc,
    void *context)
{
    void *pc = get_register(context, REG_PC);
    proc_descr_t *pd = get_procedure_descriptor(pc);
    int ra_offset = ra_slot_offset(pd);

    /* assume the return address is on the stack */
    loc->stored.type = ADDRESS;

    if(in_prologue(pd, pc)) {
        find_ra_in_stackframe_prologue(state, loc, context, pd);
    }
    else if(in_epilogue(pd, pc)) {
        loc->stored.type = loc->current.type = REGISTER;
        loc->stored.location.register = entry_register(pd);
        loc->current.location.register = entry_register(pd);
    }
    else {
        /* in the body of the procedure */
        void *sp = get_register(context, REG_SP);
        void *ra_slot = sp + ra_offset;

        loc->current.type = ADDRESS;
        loc->current.location.address = ra_slot;
        loc->stored.location.address = ra_slot;
    }
}

Figure 3.8: Pseudo-code for determining the current return address location for a stack frame function.
void
find_ra_in_stackframe_prologue(prof_state_t *state,
    struct locinfo *loc,
    void *context, proc_descr_t *pd)
{
    void *start_address = get_function_start_address(pc);
    int stack_set_offset = stack_set_offset(pd);
    int store_ra_offset = store_return_address_offset(pd);

    if(pc <= start_address + store_ra_offset) {
        void *ra_slot;

        if(pc <= start_address + stack_set_offset) {
            /* haven’t allocated the frame yet.
               compute save location */
            void *fp = get_register(context, REG_FP);
            int frame_size = frame_size(pd);
            ra_slot = fp - frame_size + ra_offset;
        }
    else {
        /* have allocated frame; haven’t stored the ra yet */
        void *fp = get_register(context, REG_FP);
        ra_slot = fp + ra_offset;
    }

    loc->current.type = REGISTER;
    loc->current.location.register = entry_register(pd);
    loc->stored.location.address = ra_slot;
}
else {
    /* allocated frame and stored ra */
    void *fp = get_register(context, REG_FP);
    void *ra_slot = fp + ra_offset;

    loc->current.type = ADDRESS;
    loc->current.location.address = \
        loc->stored.location.address = ra_slot;
}
}

Figure 3.9: Pseudo-code for determining the location of the return address when in
the middle of the prologue of a stack frame function.
### Register frame procedures

<table>
<thead>
<tr>
<th>Area</th>
<th>Modified location</th>
<th>Recorded location</th>
</tr>
</thead>
<tbody>
<tr>
<td>prologue</td>
<td>entry register</td>
<td>save register</td>
</tr>
<tr>
<td>body</td>
<td>save register</td>
<td>save register</td>
</tr>
<tr>
<td>epilogue</td>
<td>$ra</td>
<td>$ra</td>
</tr>
</tbody>
</table>

### Stack frame procedures

<table>
<thead>
<tr>
<th>Area</th>
<th>Modified location</th>
<th>Recorded location</th>
</tr>
</thead>
<tbody>
<tr>
<td>prologue, before modifying $sp</td>
<td>$ra</td>
<td>on stack</td>
</tr>
<tr>
<td>prologue, before storing $ra</td>
<td>$ra</td>
<td>on stack</td>
</tr>
<tr>
<td>prologue, after storing $ra</td>
<td>on stack</td>
<td>on stack</td>
</tr>
<tr>
<td>body</td>
<td>on stack</td>
<td>on stack</td>
</tr>
<tr>
<td>epilogue</td>
<td>$ra</td>
<td>$ra</td>
</tr>
</tbody>
</table>

Table 3.1: A summary of how the profiler inserts the trampoline in the different types of procedures that occur in programs. The “modified location” indicates the place where the return address may be found and the “recorded location” indicates the register or address stored in the profiling state for future reference.

Accurately diagnose all performance problems. Consider a sampled call path that was sampled a large number of times: such a path is known as a *hot path*. This information indicates a performance problems at the end of the call path—else it would not have been sampled so frequently—but more information is necessary to determine the nature of the performance problem. Was the function at the end of the hot path called a few times or many times? If the function was called a few times, then the focus of tuning should be that function. If instead the function was called many times, looking higher in the call chain for possible improvements would likely be a useful first step. Providing extra data about the calling patterns can help identify points in a hot path to examine for potential performance improvement.

Either the sample bit or the trampoline can be used to record extra information to aid in later analysis of the application’s performance. A count is associated with every edge in a recorded path. When using the sample bit, the last edge on the discarded suffix of the previous call path has its count incremented upon taking a new sample. When using the trampoline, the count is incremented when activating the trampoline.
for the first time after a sample. We increment only the last edge, as Arnold and Sweeney showed that incrementing all edges along the call path distorts the collected data [5]. This count represents the number of returns seen from sampled functions and can be used to discriminate between the two cases mentioned above. Therefore, using a sample bit or trampoline not only increases the efficiency of stack sampling, but provides more information in establishing the exact location of performance problems.

3.2.2 Sampling with a trampoline

Using a sample bit or trampoline to make our stack sampling procedure more efficient means that we have to modify our procedure for collecting call stack samples. Figure 3.10 shows what such a modified procedure looks like. Instead of stopping only when the program counter of the unwound context is zero, we also halt unwinding whenever the program counter of the unwound context is equal to the trampoline. This latter case means that the return address of the previous context was the trampoline and has therefore already been seen during a previous sample. Note that the procedure, as written, assumes that a trampoline is being used; the modifications are small if a sample bit is being used instead. Furthermore, the procedure collects only a suffix of the active call stack—the activation records that are new since the last sample. This suffix must be merged in some manner with the previous sample; this trivial process is not shown here.

3.3 Coping with program features

In the previous section, we described a method of profiling that will work for simple programs compiled with a naive compiler. Optimized programs, however, present several complexities that are unaccounted for in the basic design above. For example, functions in real programs do not always return to their immediate caller, nor do they always follow a particular calling sequence. Real programs can load and unload arbitrary code at runtime. In addition, several practical concerns must be addressed.
list_t *
collect_stack_trace(context_t *context)
{
    context_t copy;
    void *pc;
    memcpy(&copy, context, sizeof(context_t));

    list_t *trace = create_list();
    pc = get_register(&copy, REG_PC);

    do {
        push(pc, sample)
        unwind(&copy);
        pc = get_register(&copy, REG_PC);
    } until(pc == TRAMPOLINE_ENTRY_ADDRESS || pc == 0);

    return trace;
}

Figure 3.10: Collecting a call stack sample when the trampoline is present.

In this section, we describe these features and how they can be handled by a portable design.

3.3.1 Unsafe functions

Just because the trampoline can be inserted at any point during a program’s execution does not necessarily mean that it should.\(^3\) For instance, modifying an activation record for the trampoline to return to the trampoline will cause an infinite loop. Similarly, the profiler cannot take stack samples inside certain functions (e.g., the runtime loader). Such functions are collectively termed unsafe to indicate that samples should not be taken during their execution, nor should the trampoline be inserted.

\(^3\)The discussion which follows assumes that an asynchronous event is being monitored; the considerations under discussion only apply to such events. Synchronous events always occur at well-defined places within the code, whereas asynchronous events do not.
If the handler for the sample event determines the context is unsafe, one of two strategies can be employed. One strategy is to increment an “unsafe samples” counter and returning without taking a sample. The unsafe samples counter can be examined after the profile run has finished to provide a measure of the overhead of the profiler. A second strategy is to record samples taken in unsafe regions as a flat profiler would. Again, these samples can be analyzed to provide a measure of the overhead of the profiler, but recording the samples in this manner is more informative than mixing them all together in a single “unsafe samples” counter.

Identifying unsafe functions is not an exact science. Checking to see whether an address lies within an unsafe function is highly platform dependent, since the process depends on platform details such as whether or not libraries are relocated. Therefore, in this high-level design, we will assume the use of an “oracle” function, \texttt{address.is.unsafe.p(address)}, which returns true if the \texttt{address} parameter lies within an unsafe region of code. Once this function has been provided, it should be called whenever a sample event has been received, passing as an argument the instruction pointer of the event context. If the address is unsafe, no trampoline insertion or stack unwinding should be done. Otherwise, profiling may proceed as normal.

### 3.3.2 Dynamic linking and loading

In addition to permitting programs to link to shared libraries at compile time, modern operating systems also allow programs to load and unload shared libraries at runtime, a process known as \textit{dynamic loading}. Dynamic loading opens up the possibility that a particular address may refer to several different functions during the execution of a program. As the profiler collects program counters only during stack unwind operations, some provision must be made for mapping these program counters to their containing functions after the profile run has finished.

To facilitate this mapping, the profiler identifies each sample taken with the shared
libraries loaded at the time of the sample to unambiguously resolve addresses. We call the list of shared objects loaded at a particular time an epoch; every calling context tree collected is associated with a particular epoch. When a shared object is loaded at runtime, a new epoch is constructed. At each sample event, the handler checks to see whether the current epoch matches the epoch in which the current calling context tree was collected. If the epochs match, then the sample-taking proceeds as normal. Otherwise, a new calling context tree must be created to hold samples for the new epoch and then samples are accumulated into this new tree.

While the loading of shared objects is a convenient time to create new epochs, there is no reason that new epochs could not be created at other times. For instance, a program that was aware of the profiler’s existence could ask the profiler to begin a new epoch or resume a prior epoch at certain points during execution: an epoch associated with initialization, an epoch associated with each distinct computation phase, and so forth. The gathered samples will then be grouped by epoch automatically, rather than the analyst discerning after the fact which samples belong to which computation phase. This facility is especially useful if the profiler is recording stack samples in a tree, as epochs provide a measure of temporal grouping that the tree data structure is unable to supply.

Although dynamic linking is convenient and flexible, a small runtime cost must be paid for these features. Therefore, some applications use static linking, where all the necessary routines for program execution are collected into one monolithic binary. In principle, the design of the profiler accommodates statically linked programs if its initialization routines are called during program startup. (The initialization routines are normally called as part of the dynamic loading and unloading of the profiler library.) However, the profiler relies on being able to override several standard library functions to provide some of its functionality. These overrides are normally accomplished through dynamic linking; some other method would need to be used for statically
int
f()
{
    ...

    return g(local_var);
}

Figure 3.11: Pseudo-code showing a function with a tail call.

linked programs.

A simple method for accommodating statically linked programs is to link the program’s object files with the profiler library such that the program calls the profiler’s versions of overridden functions. The original functions (from the libraries which are also being statically linked with the program) need to be renamed to not conflict with the profiler’s versions. Finally, the profiler’s overridden functions should be linked against the renamed library functions. (This last linking step is accomplished by function pointer capturing in a dynamically linked scenario.)

3.3.3 Optimized calls

Most functions will return to their callers via a “jump-to-register” instruction. The trampoline-as-sentinel operation depends crucially on this property. Some functions, however, do not return via a “jump-to-register” instruction. Figure 3.11 shows a function \( f \) that returns a value computed by function \( g \). Notice that there is no work which needs to be done after \( g \) returns other than returning the value \( g \) computed. \( g \) is said to be in \textit{tail position}; the call to \( g \) is a \textit{tail call}. If the compiler recognizes this, a more efficient calling sequence can be generated for the call to \( g \).

A “dumb” compiler might compile the call to \( g \) and subsequent return from \( f \) as follows:

1. Save caller-saved registers;
2. Call g;
3. Restore caller-saved registers;
4. Restore callee-saved registers for f;
5. Return from f.

Steps 1 and 3 may not be necessary, depending on the cleverness of the compiler. However, a smarter compiler which recognizes that the call to g is a tail call would generate something similar to the following code:

1. Restore callee-saved registers for f;
2. Jump to g. g will now return its value directory to f’s caller without returning first to f.

This sequence is obviously much more efficient than the previous sequence. However, a compiler which does this tail-call optimization can confuse the profiler. Assume that f’s activation record has been modified to return through the trampoline. In the first scenario given above, f would indeed return through the trampoline. In the second scenario, however, g would be returning through the trampoline (since the trampoline is f’s “caller”). The profiler is not aware of this situation and it is possible for the profiler to record inaccurate data in this case.

Three possibilities exist: g will return without a sample event occurring, a sample event will occur in g, or a sample event will occur in a procedure directly or indirectly invoked by g. The first one is simpler to explain and to handle. If g returns before a sample event occurs, the trampoline will be activated. As the trampoline adjusts the cached call stack to reflect the current state of the program’s call stack, a return will be attributed to f (since f’s node was never popped off in the first place). This is exactly what would happen if the return from f was a “normal” return and not a tail call. No data is lost nor are peculiar results observed.
The more complex cases occur when a sample event occurs in \( g \) or in a procedure directly or indirectly invoked by \( g \). As we will see, these two cases can be handled in the same way and we will refer to them as a combined case throughout the rest of this discussion. In this case, the profiler must be able to detect the tail call and recover gracefully. Removing the trampoline will be impossible, since the profiler does not know where the trampoline is (\( g \) will have probably saved its return address in an entirely different location than \( f \)). Furthermore, inserting another trampoline would violate the profiler's invariant of at most one trampoline per thread of control.

This case will be handling during stack unwinding, as tail calls make it necessary to perform stack unwinding slightly differently anyway. When tail calls are not accounted for, the profiler can simply unwind the stack until a frame where the instruction pointer equals the trampoline's entry address is located, as shown in Figure 3.10. The profiler then knows to halt the unwinding process and to append the collected stack trace to the cached call stack. In the presence of tail calls, however, a frame whose return address is the trampoline's entry point is not necessarily located at the end of the profiler's cached call stack. Therefore, in the presence of tail calls, the profiler must ensure, during unwinding, that the cached call stack is maintained as a prefix of the program's actual call stack.

Properly detecting and handling tail calls implies changing the unwinding process to accommodate the necessary checks. We will use procedure descriptors to halt the unwinding process rather than the presence of the trampoline. Doing this assumes each procedure has a unique procedure descriptor. Prior to unwinding, the procedure descriptor for the procedure \( p \) at the top of the cached call stack is retrieved; call this procedure descriptor \( pd \). At each visited call stack frame, then, the procedure descriptor for that call frame is retrieved and compared with \( pd \). If the procedure descriptors are different, then the unwinding should continue. If the procedure descriptors match, the frame pointer from the bottom of the cached call stack is compared with the frame
pointer for the current frame. (Note that both frame pointers have been canonicalized; see section 3.3.4.) If the frame pointers are different, unwinding should continue. If the frame pointers are equal, then unwinding should halt. However, the instruction pointers need to be compared to determine where in the cached call stack the new collected call stack should be spliced. Different instruction pointers indicate different children in the CCT and require popping the last node on the cached call stack before attaching the collected suffix. Identical instruction pointers do not require any nodes to be removed from the cached call stack before attaching the collected suffix.

Handling tail calls with this framework proceeds as follows. When a sample event occurs, the trampoline is removed from its prior location; this is the reason the trampoline insertion computed two locations when the trampoline was being inserted. If it cannot be removed (i.e. the place where it should have resided contains different data), then the profiler knows that a tail call has taken place. If during the unwinding process, the current frame’s instruction pointer is found to be the entry point of the trampoline, then the tail-called procedure has been located—the frame prior to the current one contained the trampoline.

Several actions then need to occur. First, the stale pointer to the trampoline is spliced out; the profiler can compute where it must be from the procedure descriptor of the previous frame. Second, the topmost node on the cached call stack needs to be removed, since its associated function no longer appears in the active call chain (this node corresponds to f in the above example). Third, the target procedure descriptor needs to be updated from the new top frame in the cached call chain; the profiler knows this function must still exist in the active call chain. After taking these steps, the unwinding process will continue for one more unwind of the call stack (finding the caller of f in the running example).

---

4Since the procedure descriptors match and the frame pointers are different, this case represents a recursive call to p.
The only oddity recorded is that \( f \)'s caller, call it \( e \), will call function \( g \) in the collected call graph, even though this call does not occur textually in the program. This is simply a consequence of tail call optimization and not the fault of the profiler, since there is no way to determine at runtime that the call chain was actually \( e \) calls \( f \) calls \( g \). Debuggers face the same difficulty in displaying stack frames in the presence of tail calls.

A similar issue occurs with programs that contain recursive tail calls. Depending on how sophisticated the compiler is, these calls may compile to “ordinary” procedure calls or they may be optimized into loops. In the former case, the profiler will treat such calls as it would any other call, i.e. they will be correctly recorded. In the latter case, samples will be correctly attributed to the function, but there will be no way of discerning which recursive invocations of the function incurred the most samples. Again, debuggers face the same difficulty with representing recursive tail calls and users have indicated the performance gain from tail calls are preferable to easily debuggable programs [34].

### 3.3.4 Frame pointer canonicalization

The profiler must be careful to provide an intuitive picture of the program’s behavior to the user. Consider the following scenario: a function \( f \), which allocates stack space, is sampled in two different regions, once in its prologue and once in its body. From the point of view of the profiler, these are distinct nodes in the calling context tree, since their instruction pointers and their frame pointers are different. During the analysis phase, however, the user will probably wish to see these nodes merged in some way, as they are really part of the same function (the difference in their frame pointers being irrelevant).

While this situation could be handled in the analysis phase, it is somewhat easier to handle it during the stack sampling phase of the profiler. This handling is done by canonicalizing the frame pointer: ensuring that the frame pointer for any point in the
function is always recorded as its value upon entrance to the function. The procedure descriptor for each function indicates its (fixed) frame size, so this canonicalization is easy.

Performing the canonicalization is straightforward, but as when inserting the trampoline, several different cases must be considered. In the subsequent discussion and code samples, we assume that the machine’s stack grows downward. Given a context, if the context’s instruction pointer lies within a register frame procedure, computing the canonicalized frame pointer is trivial. Otherwise, on the first unwind, the profiler must determine if the stack frame procedure associated with the context’s instruction pointer is in its prologue, body, or epilogue. If the instruction pointer points into the prologue, the profiler must decide if the function’s activation record has been allocated yet. If instead the instruction pointer is contained within the body, the profiler adds back the frame size to find the canonical frame pointer. Finally, if the procedure is in its epilogue, computing the canonical frame pointer is also trivial. For procedures below the top of the call stack, the instruction pointer is always located in the body and the profiler need only add back the frame size to canonicalize the frame pointer. Figures 3.12 and 3.13 show pseudo-code detailing this process; the given pseudo-code would be part of a procedure which collects the new suffix of the current call stack.

The value of canon_frame_pointer will then be used for the appropriate CCT node instead of the frame pointer of the context. This ensures the user sees a consistent picture at all times.

3.3.5 Overriding library calls

Running the profiler in the same address space as the application allows the application to (unknowingly) shut off the profiler. For instance, the application might install its own signal handler for the signal that notifies the profiler of a sample event. This installation would replace the handler installed by the profiler and ruin the profiling
void **canon_frame_pointer;
/* 'first_unwind' is true if before the first unwind of the call
stack; if we have not yet performed an unwind, then we do not
know where in the procedure we are (prologue/body/epilogue)
and must do some computation to discover our frame pointer
upon procedure entry. */
int first_unwind;
/* the procedure descriptor of the current context's function during
each unwind step. */
proc_descr_t *pd;
/* provided from outside the procedure */
context_t *ctx;

...

void *frame_pointer = get_register(ctx, REG_FP);

/* place the canonicalized frame pointer in 'canon_frame_pointer' */
if(is_register_frame(pd)) {
    canon_frame_pointer = frame_pointer;
} else if(first_unwind) {
    canon_frame_pointer = \
    canonicalize_stack_frame_pointer(ctx, frame_pointer, pd);
} else {
    /* the frame will always be allocated in subsequent unwinds */
    unsigned int frame_size = frame_size(pd);
    canon_frame_pointer = frame_pointer + frame_size;
}

Figure 3.12: C-like pseudo-code to describe the frame canonicalization process.
void *
canonicalize_stack_frame_pointer(context_t *ctx, void *frame_pointer,
    proc_descr_t *pd)
{
    void *ip = get_register(ctx, REG_PC)
    void **canon_frame_pointer;

    if(in_prologue(pd, ip)) {
        if(past_frame_allocation_point(ip, pd)) {
            /* add back the stack frame */
            unsigned int frame_size = frame_size(pd);
            canon_frame_pointer = frame_pointer + frame_size;
        }
        else {
            /* frame has not yet been allocated */
            canon_frame_pointer = frame_pointer;
        }
    }
    else if(in_epilogue(pd, ip)) {
        canon_frame_pointer = frame_pointer;
    }
    else {
        /* must be in the body */
        unsigned int frame_size = frame_size(pd);
        canon_frame_pointer = frame_pointer + frame_size;
    }

    return canon_frame_pointer;
}

Figure 3.13: C-like pseudo-code for canonicalizing the frame pointer of a stack frame procedure.
run. To guard against such application actions, we override several functions in the standard C library with functions of the same name in the profiler’s library. Since the application will consult the profiler’s library for symbols before it consults the system C library, dynamic linking will cause the application use the profiler’s functions rather than those in the C library.

Certain calls, however, are not dangerous and should be allowed to complete—the application may wish to install a handler for SIGSEGV—and therefore the original functions must also be available to call. During the profiler’s initialization, function pointers to the original library functions are captured. The profiler’s implementations can then apply checks to the arguments and call the “real” functions if the arguments are “good”; otherwise, success is returned without actually doing anything. As an example, the profiler’s version of setitimer checks to see whether the ITIMER_PROF timer is being set. If so, then no action is taken⁵; otherwise, the arguments are passed to the C library function via the function pointer. If the profiler itself needs to call any overridden functions, it uses the captured function pointers directly.

Certain library calls that serve as “hooks”—specific actions the application performs of which the profiler would like to be notified—are also overridden. One example would be overriding dlopen to learn when a new profiling epoch should begin. In the profiler’s version of dlopen, the original dlopen function is called to load the library and then the runtime loader is queried to determine the address at which the new library was loaded. While the querying of the runtime loader will differ from platform to platform, the important thing to discover is the address where the text section of the newly loaded library has been placed; this address should be recorded in the newly created epoch. This address will be used after the application exits to assist in mapping from the instruction addresses recorded during sampling to function names.

⁵To ensure that the profiles collected are consistent, no action is taken. However, since the profiler is modifying the semantics of the program by not performing the action, an equally valid action would be to halt profiling and allow the program to continue normally.
3.3.6 Non-local exits

Real programs do not always follow an orderly sequence of calls and returns. A non-local exit occurs when a function \( f \) transfers control to a function \( g \) that occurs above \( f \) in the call stack, regardless of whether \( g \) was the immediate caller of \( f \). Non-local exits are commonly found in programs that use exception handling; exceptions thrown in functions can be handled by a call-chain ancestor and the intermediate calling contexts are never returned to via a normal return. The standard C functions \texttt{longjmp} and \texttt{setjmp} are often. \texttt{setjmp} stores its calling context into a user-supplied variable; \texttt{longjmp} accepts a context captured in this fashion and returns to it.

Non-local exits must receive special attention from the profiler to ensure the profiler records accurate data when non-local exits are used. As an example of why non-local exits must be treated specially, consider Figure 3.14, part \( a \) where function \texttt{encrypt} is set up to return through the trampoline to its caller, \texttt{commit}. The stack on the right of part \( a \) is the profiler’s cached call stack, which is a prefix of the active call stack. \texttt{encrypt} now returns through \texttt{longjmp} to its caller. Part \( b \), on the left side of Figure 3.14 shows the situation, where the trampoline has been removed and the profiler’s cached call stack is no longer a prefix of the program’s call stack. Therefore, the profiler must be notified when a non-local exit takes place, so that the trampoline can be moved appropriately and its data structures updated.

While \texttt{setjmp} and \texttt{longjmp} can be used to implement a coroutine-based user-level threading system, we do not handle the complexity posed by that situation. Instead, the profiler only handles the simpler, “exception-handling”-like cases. If a threading system of any sort is being used in the program, the added complexity posed by that case is handled by a separate thread-aware profiler (see section 3.3.7).

We override \texttt{longjmp} to perform additional handling before returning to the given context. The basis of our additional handling is to check whether the \texttt{longjmp} is jumping into the “middle” of the cached call stack. If this is the case, then frames
Figure 3.14: Hazards of `longjmp`. In part (a), the program and profiler are running normally; the shaded part of the diagram indicates the cached call stack of the profiler. In part (b), `encrypt` has returned through `longjmp`, invalidating the profiler's data structures.
need to be popped off the cached call stack to maintain it as a prefix of the active call stack. Our extra handling retrieves the frame pointer \( f_p_r \) from the context to be restored, canonicalizes it (see section 3.3.4), and compares it with the frame pointer at the top of the cached call stack, \( f_p_c \). This effectively compares the two call frames. If the context being restored lies above the frame in which the trampoline is installed, then nothing needs to be done. Similarly, if the context being restored already has the trampoline installed, then nothing needs to be done.

The more interesting case occurs when the context being restored lies below the frame in which the trampoline is installed. In this case, the trampoline must be moved and the cached call stack must be modified. We pop nodes off the cached call stack until we find a node whose frame pointer is equal to the canonicalized frame pointer we computed previously. (Such a frame must exist due to restrictions on the manner in which \texttt{setjmp} and \texttt{longjmp} can be used.) The context that is the target of the non-local exit is then altered to return through the trampoline. In effect, this process emulates what would have happened if the non-local exit had not occurred and functions being passed over had simply returned normally. After this handling, the standard \texttt{longjmp} can be called with the (possibly altered) context and execution proceeds as normal.

As a practical matter, there are actually three different versions of the \texttt{longjmp} function, each with its own corresponding \texttt{setjmp}: \texttt{longjmp}; \_\texttt{longjmp}; and \texttt{siglongjmp}. Each \texttt{longjmp} function must be overridden separately. Much of the code necessary for the approach above can be shared between the three functions.

Exceptions can be handled in much the same manner. We assume that exceptions follow the “zero-overhead” or “two-phase” model [16]. This model is commonly used in C++ and Java implementations. In this style of exception handling, code regions (such as functions, but potentially more fine-grained, as a function may have several blocks in which exceptions are caught and processed) are associated with an handler
for a particular type of exception. When an exception is thrown, the system searches up the call chain from the point of the thrown exception to locate a handler able to process the exception. If no handler is found, the program exits. Otherwise, the stack is unwound to the function that encloses the handler and control is transferred to the handler.

Once the search for an appropriate handler is successful, handling exceptions reduces to handling `longjmp`, as the transfer of control to the handler is simply a non-local exit. We require that there exist a function which is called to initiate the search process described above.\(^6\) This function is overridden to serve as a "hook" whereby the profiler may be notified that an exception is being processed. When notified of such an event, a flag is set signifying that an exception is being processed and no stack samples should be taken, nor should the trampoline be moved. There must also exist another function which performs the transfer of control to the handler (this function shares many features with `longjmp`). We override this function as well, clearing the "handling exception" flag before resuming control at the handler.

### 3.3.7 Multithreading

Dealing with multithreading is straightforward if the sample event source is capable of handling multiple threads. Some hardware performance counter implementations, for example, may not count events per-thread, but rather per-process, which makes them unreliable as a sample source for multithreaded programs. Similarly, wall-clock timers are acceptable only if they provide a way of signaling multiple threads at once. On the other hand, synchronous sample events are nearly always acceptable, as they will automatically be isolated to the thread triggering the event. The profiler must be notified of thread creation to allocate a profiler state for that thread. Each thread then possesses its own trampoline and the sampling process does not require inter-thread communication.

\(^6\)On Tru64/Alpha, this function is `exc.dispatch.exception`. 
In the standard pthreads threading interface, thread creation is detected by over-
riding the function pthread_create. pthread_create takes several arguments; the
important ones for this discussion are a function of one argument for the newly cre-
ated thread to execute and an argument to pass to that function. The function and
its associated argument are placed into a trampoline structure by the profiler’s version
of pthread_create. A thread trampoline and this structure are then passed to the
actual pthread_create function. Note that the thread trampoline is different from
the trampoline used in the context of stack sampling.

This trick is done to ensure that the profiler state is initialized in the context
of the newly created thread, since threads are not permitted to access thread-local
data of other threads. The thread trampoline enables the profiler to gain control
after thread creation, initialize the thread’s profile state, and store the profile state
in a thread-local data slot. Once the profiler state has been initialized, the actual
application function and its associated argument are retrieved from the trampoline
structure. The application function is then called with the proper argument. After
this function returns via pthread_exit, control returns to the thread trampoline,
which then writes out the collected profile and destroys the thread.

Unfortunately, not all created threads call pthread_exit, instead relying on ap-
plication exit to clean up any threads which have been created. pthread_exit would
be an appropriate place to write out the dying thread’s profile data. But since
pthread_exit is not guaranteed to be called, the profiler must find another “hook”
at which to write out the profile data.

There are several appropriate places at which the profile data can be written to
disk; one technique common to all of them is maintaining a list of threads which
have been created. Nodes are added to this list by the thread trampoline using a
thread-safe mechanism to ensure no nodes are lost. Each node contains a reference to
the profiler state for that particular thread. Maintaining this list is necessary because
no one thread can be relied upon to do the writing, as the profiler state is thread-
local and threads may not access other threads' thread-local data in \texttt{pthreads}. A
time is then chosen when the profiler can be sure that no more threads can be created
(library finalization is a good choice). At the chosen time, the list can then be iterated
through and each profiler state written out to disk.

3.4 Summary

This chapter has described the design of a call stack sampling profiler that uses pro-
cedure descriptors to facilitate efficient unwinding. This efficiency is achieved by
modifying the frames of sampled functions to include a trampoline; this trampoline
then serves to distinguish between call frames that were seen during previous sample
events and fresh call frames. In addition to this improvement to naive stack unwind-
ing, this section also described how to address several practical concerns and how the
profiler copes with real-world program features and optimizing compilers.
Chapter 4

Alpha/Tru64 implementation

Even a well thought-out portable design may require a non-trivial amount of work when the time comes to implement the design. Creating a software artifact from a design also validates the design by proving that the design can be implemented cleanly. In this section, an implementation for the Tru64/Alpha platform of the high-level, portable design from the previous section, csprof, is described. Details of the particular methods and functions used for sampling and unwinding are given. Making csprof work on code from a real-world compiler also turned up several issues; we identify problems that cropped up during development and the workarounds for those problems.

4.1 Sampling and unwinding details

Tru64 provides a well-specified interface for procedure descriptors in the library libexec. Procedure descriptors are split into two separate parts: code range descriptors (CRDs) and runtime procedure descriptors (RPDs). A single procedure may have may distinct code ranges, but only one runtime procedure descriptor. The code range descriptor identifies features of a range of addresses, such as whether the region contains a prologue region, if the region uses stack allocation that needs de-allocation, and if the region serves as an exception handler. CRDs can be used to help determine if an address lies within an epilogue region, e.g. if an RPD indicates the address's procedure uses the stack, but the CRD indicates the stack has already been deallocated (perhaps because the instruction at the address is a return instruction). RPDs contain the necessary functionality described in section 3.2.1 for inserting the tram-
pline. During the rest of the chapter, “procedure descriptor” will be understood to refer to the appropriate functionality in either CRDs or RPDs.

Tru64 actually provides three kinds of RPDs: RPDs for stack frame procedures, RPDs for register frame procedures, and RPDs for null frame procedures. The latter is a special kind of register frame RPD with default values for all fields and is represented by the C value NULL. This special RPD creates problems while unwinding, as it violates the assumption that different procedures have distinct RPDs. However, CRDs for null frame procedures are unique; in the special case of a null frame procedure encountered during unwinding, the CRD of the bottommost frame on the cached call stack is used instead of the RPD for equality comparison.

Compaq’s libexc library also provides a stack-unwinding interface and exception handling facilities. Stack unwinding is done via the function exc_virtual_unwind. This function takes a context argument and modifies the context to point to the previous frame in the call chain. Exceptions are handled by calling exc_dispatch_exception, which searches for an appropriate exception handler. csprof overrides this function to be notified of when exception handling processing is taking place to avoid taking samples during that time. After an appropriate exception handler is found, exc_continue is called to transfer control to the handler; csprof overrides this function as well to perform the necessary handling for non-local exits and to indicate that sample taking is again possible.

csprof monitors the wall clock time of the application being profiled by using the ITIMER_PROF interval timer provided by the kernel. This timer counts down while the process is executing and while the kernel executes on behalf of the process. Interval timers can be configured to be “one-shot” timers or to begin counting down again after expiring. csprof uses the one-shot approach, resetting the timer upon exit from the signal handler to minimize overhead. A SIGPROF signal is sent when the interval timer expires; csprof catches this signal and executes the appropriate sample-taking code.
An alternate asynchronous sample source available on some platforms is the overflow of hardware performance counters. Tru64's kernel does not expose hardware performance counter overflow to user programs, so we were unable to use this sample source. Tru64 provides an interface to query the runtime loader about which shared objects are currently loaded; we used this interface to implement epoch support. Discovering the unsafe functions was done by trial and error. Samples are stored in a tree structure, as discussed in section 3.1.2; the cached call stack contains pointers into the tree for faster insertion of the call stack suffixes collected during unwinding.

### 4.1.1 Unsafe function detection

In section 3.3.1, we discussed the necessity of marking certain regions of code as "unsafe", regions where the trampoline may not be inserted and stack sampling cannot occur. Since we are using an asynchronous sample event source, we needed to implement safe checking in csprof. As stated in section 3.3.1, determining which functions or regions of code are unsafe is not an exact science. We did so mostly by trial and error, testing csprof on a wide range of code. The best description for the regions of code we identified during this trial and error process are "tricky" regions: those which perform operations which are difficult to express with procedure descriptors, thus making trampoline insertion and unwinding difficult while those regions are executing. The setjmp family of functions falls into this category, as program-terminating functions such as exit and abort.

Once the relevant regions were identified, performing the checking was a simple matter. address.is.unsafe.p uses two arrays, one of the sorted "start" addresses of unsafe functions and another of the sorted "end" addresses of unsafe functions. To discover whether a particular context is unsafe, a binary search is made on the "start" array to find the largest address less than the program counter of the context. The corresponding entry in the "end" array is then compared against the program counter; if the program counter is less than or equal to the "end" entry, then the
region is unsafe. Otherwise, the sample-taking and trampoline insertion proceed.

Calculating the contents of the “end” array cannot be done at runtime, as there is no way to determine the last instruction of a function. Searching forward from the beginning for a return instruction does not work in general because of tail calls and/or basic block reordering. Functions may also contain multiple exit points. Therefore, for most unsafe regions, the end of the region is hard coded into the profiling library–the start of the region is hard coded as well for convenience’s sake.

This decision makes two assumptions: the first is that the regions which those addresses represent will always be mapped at those virtual addresses. Since the unsafe regions tend to be in crucial system-provided infrastructure, such as the runtime loader or the C library, and that Tru64 assigns system libraries a preferred load address which is unlikely to change, this is a reasonable assumption. The second is that the environment in which the library is used will not change–new libraries are not used, new loaders are not installed, and so forth. This assumption is more dangerous to make; one of the primary promises of dynamic linking is that updating a library will (in general) not cause applications that rely on that library to cease working. System upgrades should be as transparent as possible to system applications.

However, this second assumption is reasonable in the scientific computing community, where upgrades to a running system tend to be few and far between. The profiler can also check whether the currently installed system libraries correspond to the libraries with which it was compiled; if a mismatch is detected, the profiler will exit with an error message instructing the user to recompile the profiling library.

### 4.2 Implementation difficulties

Two features of Tru64 made the implementation more complex than the high level design presented in Chapter 3: the procedure descriptors were not nearly as detailed as the designed-for procedure descriptors and the compiler did not always follow the platform’s calling conventions at high levels of optimization. These are the sort of
details that make implementation tricky, and so they are covered below.

4.2.1 Marking activation records

The description in section 3.2.1 of how to insert the trampoline relied on being able to make a clear distinction between body and epilogue code. Making this distinction is especially important for stack frame procedures that contain call-free paths and therefore do not need to reload the return address from the stack frame. For these procedures, the entirety of such paths are epilogue code as far as csprof is concerned. Unfortunately, Tru64 procedure descriptors do not provide enough information to conclusively determine if the procedure is in its body or epilogue.

To deal with insufficient body/epilogue distinction and clever compilers, csprof uses a heuristic approach to inserting the trampoline in a stack frame procedure. Once csprof determines the procedure is not executing its prologue, it computes the location of the return address on the stack. The contents of this location are then retrieved. So far, nothing different from the body case has happened. To detect whether the procedure is executing its epilogue or a path along which the return address is not reloaded from the stack, the profiler scans forward several instructions in the instruction stream. This scanning is similar to how calling convention violations are handled during stack unwinding (see section 4.2.2).

If an instruction that reloads the return address from the stack is found, then the insertion process stops: csprof knows that modifying the stack is sufficient. Otherwise, if a return instruction or an instruction modifying the frame pointer is found, then csprof can be confident the procedure is in its epilogue. In this case, the context is modified to contain the trampoline. If none of the above conditions are met or a branch instruction is found, csprof assumes the procedure is executing its body and replaces the return address on the stack.

To account for the possibility of being on a path through the procedure that does not reload the return address, csprof makes one additional check. We will call this
the "return address equivalence check" for convenience. The return address retrieved from the stack is compared with the return address register in the provided context. If they are equal, then csprof assumes the return address may not be reloaded from the stack and replaces the return address in the context. This modification introduces two dangers, both related to corruption of the user program's state: the program's access to global data may be affected and the data value in the register may not have been a return address. The second concern is unlikely to arise in practice, as a piece of user data is unlikely to be exactly equal to the return address of a procedure.

The first concern, however, is very real, and requires some background discussion. Global variables in Tru64 are accessed using a dedicated register, $gp ("global pointer"), that points to the global storage area for a particular region of code. This pointer is initialized upon entrance to a procedure using the entry address of the procedure. After calling other procedures, which may use different global pointers, there needs to be some mechanism to "reset" the global pointer. The Alpha calling conventions specify that the return address in $ra will be used to reinitialize $gp after a function call. Two paired instructions, a 1da and a 1dah are used to add the appropriate offset. These instructions should occur immediately after a function-calling instruction such as jsr is used.

However, if the trampoline is inserted before the global pointer is reset, then the global pointer will be initialized based on the address of the trampoline—which most likely points to garbage from the perspective of the application. This mistake causes incorrect data to be referenced by the application or, more often, invalid memory references and program crashes. In addition, when using optimization, the Compaq system compilers for Tru64/Alpha do not always place the 1da/1dah pair immediately following a call instruction, making it difficult for csprof to discern if inserting the trampoline will cause problems with the global pointer.
As noted previously, some stack frame procedures, while they save their return address on the stack, do not always reload the return address from the stack before returning. Hence the return address equivalence check when inserting the trampoline. But if the current context represents a self-recursive call, then this check will always be true, but $ra does not need to be modified. Modifying $ra before the global pointer has been computed will lead to the computation of an invalid global pointer.

To prevent computation of an invalid global pointer, yet another check is made when inserting the trampoline upon reception of a sample event. csprof examines a window of instructions forward the context’s instruction pointer to determine if any of them read from $ra. If one of them does and is not a return instruction, csprof will not modify $ra, as the global pointer has not yet been restored. If none of them do, then csprof overwrites $ra with the address of the trampoline. While it is possible that the window of instructions is not large enough or that this modification could corrupt user data, neither concern has been realized in practice.

4.2.2 Calling convention violations

csprof uses exc.virtual.unwind to handle the task of unwinding the stack to collect stack samples. Doing this proved to be easier than writing a special unwinder just for csprof. However, given that signals can occur at any point in the code, the ability to take stack samples at any point in the code is a requirement. Taking stack samples at any point in the code implies that the stack must be able to be unwound at any point in the code. At first glance, exc.virtual.unwind appeared to be up to the task.

Unfortunately, this is only true if the code region being interrupted obeys the Alpha calling conventions, which exc.virtual.unwind assumes. This assumption is perfectly reasonable, since exc.virtual.unwind was intended to be used for exception handling purposes. When handling an exception, all the currently active procedure activations are at well-defined points in their code. However, while developing csprof, several instances of calling convention violations were observed, both in system library
code and code generated by the system compiler. Since `exc_virtual_unwind` will not cope with the violations, the violations must be handled by `csprof` as special cases.

Two common violations of the calling conventions by the supplied version of the system compiler were discovered during the development of `csprof`. The first concerns the requirements placed on procedure prologues. During procedure prologues, callee-save registers are not permitted to be used as scratch registers, even after they have been saved on the stack. This stipulation enables `exc_virtual_unwind` to easily determine the location of the return address. However, the compiler will occasionally generate code that uses `$ra` as a scratch register—after saving it in the stack frame and while the prologue is still executing according to the function's procedure descriptor. If `exc_virtual_unwind` is called in such a situation—when `$ra` has been saved to the stack and then modified along with the instruction pointer still being in the prologue—then it will assume the current contents of `$ra` represent the return address. This assumption is incorrect and will, at the least, cause incorrect information to be gathered by `csprof`. At worst, `exc_virtual_unwind` will abort and terminate the program being profiled. Neither of these results are desirable.

Repairing this violation occurs during the unwinding process. Prior to the first call to `exc_virtual_unwind`, `csprof` checks whether the instruction pointer of the supplied (interrupted) context lies in a prologue region and is part of a stack frame procedure. If both of these conditions are met, `csprof` then determines whether the instruction pointer is past the point at which the return address register is saved to the stack frame. If not, then no calling convention violation is possible. Otherwise, `csprof` retrieves the return address from the stack frame and compares it to the current contents of the return address register in the context. If they match, then there is no violation. Otherwise, the value from the saved stack slot is stored into the context’s return address register to provide `exc_virtual_unwind` with a “proper” context.
Another calling convention violation concerns calling `exc.virtual.unwind` from procedure epilogues. The calling conventions state that a procedure which uses stack space must return via a two-instruction sequence wherein the first instruction deallocates the stack frame and the second instruction returns to the calling procedure. Making this restriction enables `exc.virtual.unwind`, once it has determined the instruction pointer lies outside a prologue region, to easily check whether the stack frame has been deallocated: is the current instruction a return instruction? However, presumably for efficiency reasons, the compiler sometimes separates these paired instructions (we have observed over 50 intervening instructions in extreme cases). If the context passed to `exc.virtual.unwind` has an instruction pointer that points into one of these convention-violating regions, `exc.virtual.unwind` will deallocate the stack twice, with disastrous results.

The fix for this again takes place prior to the first call to `exc.virtual.unwind`. If the instruction pointer of the context is inside a procedure that allocates space on the stack, `csprof` scans forward from the context’s instruction pointer, attempting to locate a return instruction. Locating a return instruction prior to locating an instruction that deallocates the stack space would indicate a calling convention violation. If an instruction deallocating the stack space is found, the search terminates and nothing is done. If a branch instruction is encountered, the search terminates. This choice is unsafe, as there may be a return instruction unpaired with the stack frame deallocation instruction beyond the branch. However, this unsafe choice has not been a problem in practice. If a return instruction is identified, the context’s instruction pointer is altered to point at the return instruction. The unwinder then correctly determines that the stack space has been deallocated. Note that the instruction pointer recorded in `csprof`’s CCT is the original one from the context; the alteration is done to pass a “safe” context to `exc.virtual.unwind`.

Both of these calling convention violation fixups are not necessary for every call to `exc.virtual.unwind`. They are only necessary before the first call, as the last
function in the call chain (the function that was interrupted by a sample event) is in an “unknown” state. Every subsequent unwind will be unwinding from a function in a known state—executing its body code.

4.2.3 Procedure descriptor bugs

Bugs with regard to the calling conventions in the generated code were not the only bugs discovered with the Compaq compilers. As previously mentioned, procedure descriptors are divided into two parts, RPDs and CRDs. CRDs come in several types; the relevant type for this discussion is the CRD_TYPE_NON_CONTEXT type, which specifies that the code range in question does not need to deallocate stack allocation. However, the Compaq system compilers erroneously generate these code ranges for procedure epilogues that reload registers from a stack frame, reset the stack pointer, and return. Furthermore, the unwinder assumes that if the instruction pointer falls within a CRD_TYPE_NON_CONTEXT region, then the return address is located in the return address register—an assumption that may not be true if the return address has not been reloaded from the stack.

To deal with this compiler bug, two additional checks are made while scanning for calling convention violations. If an instruction that reloads the return address from the stack is found, the RPD and CRD for that instruction are fetched. If the CRD c for the instruction is of type CRD_TYPE_NON_CONTEXT and the CRD for the original instruction pointer from the context is identical to c, this is a bug. The return address is manually reloaded from memory and stored into the context before unwinding. Additionally, the context’s frame pointer is reset, as the region is not supposed to use the stack. The second check is similar to the first; if a reset of the frame pointer is found, the context’s frame pointer is reset. Appropriate actions are also taken while inserting the trampoline.

A second compiler bug with regard to generated proper procedure descriptors was also observed. The compiler would sometimes correctly identify a small region of
code as `CRLD_TYPE_NON_CONTEXT`—usually a procedure epilogue that was moved into the “middle” of the procedure—but allow the code region to extend well beyond that small region of code. As interpreted by the unwinding code, this extension designated a large part of the function as not requiring stack deallocation during unwinding—when in fact that part of the function did use the stack. This bug would result in incorrect profiles being collected or even program crashes. As we did not have access to the system compiler’s source code, we could not fix this bug. This bug was only seen on the gcc benchmark from our experiments, which are described in the next chapter.
Chapter 5

Experimental results

Performance testing is an essential part of validating the design for a profiler: seeing if the profiler delivers the desired data with an acceptable amount of overhead. This chapter describes experimental results from using csprof on several benchmarks.

We evaluate csprof along two different axes. First, we compare the overhead of csprof with an instrumentation-based profiler. This comparison is intended to evaluate the monitoring overhead of csprof’s call stack sampling strategy. However, a profiler that simply has low overhead is not necessarily a good profiler—the null profiler which records no data about the target program has zero overhead, for example.

This observation suggests a second axis of evaluation: the accuracy of csprof as compared to other profilers. Comparing accuracy determines whether a profiler delivers appropriate “bang for the buck”: if profiler A has lower overhead than profiler B and reports information of comparable accuracy, then, all other things being equal, the use of profiler A would be preferred. We compare the accuracy of csprof against two profilers: a low-overhead flat profiler and an instrumentation-based call graph profiler. The instrumentation-based call graph profiler was chosen to illustrate that csprof not only delivers low overhead, but also profiles of comparable accuracy. By comparing csprof’s accuracy versus the low-overhead flat profiler, we aim to show that csprof’s profiles are nearly as accurate as those collected by flat profilers.

5.1 General setup

For our tests, we used the SPEC CPU 2000 benchmarks [12]. Our experimental platform was a quad-processor Compaq ES40 machine; each processor was a 21264
Alpha EV67 running at 667MHz. The machine was equipped with 2GB of RAM and was running Tru64 5.1A (rev 1885). Table 5.1 shows the compilation options used with each compiler. The altered Fortran compiler command line compared to the C and C++ compiler command lines was used to enabled high-level loop optimizations (-05); the remaining differences are command line options automatically enabled with -fast, but not with -05. To test call graph profiling with compiler-based instrumentation, the flag -pg was added to the compilation command lines, instructing the compiler to add appropriate instrumentation to the generated executable. For brevity’s sake, we refer to the compiler-based instrumentation as gprof throughout the remainder of this chapter.\footnote{Tru64 also comes with a command-line utility named hiprof to add instrumentation after linking the executable. When using this tool on the executables generated with -fast, we were unable to obtain functioning executables.} The gprof instrumentation samples the program counter 1000 times a second. We could find no way to modify this setting, so for our initial head-to-head comparison, we configured csprof to use the same sampling frequency.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Compilation flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compaq cc</td>
<td>6.4-009</td>
<td>-fast -arch host</td>
</tr>
<tr>
<td>Compaq cxx</td>
<td>6.5-014</td>
<td>-fast -arch host</td>
</tr>
<tr>
<td>Compaq f90</td>
<td>5.5-1877</td>
<td>-05 -align dcommons</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-assume noaccuracy_sensitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-math_library fast -align sequence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-assume bigarrays -assume nosize</td>
</tr>
</tbody>
</table>

Table 5.1: Compilation options for our experiments.

csprof is dependent upon the compiler for correct procedure descriptors. However, the version of the C compiler we used generates incorrect procedure descriptors for the benchmark gcc. As a result, we were unable to obtain csprof profiles for gcc.
We expect that if correct procedure descriptors were generated for gcc that profiling it with csprof would give similar numbers to those shown in the tables below.

5.2 Comparing the overhead of profile collection

Table 5.2 summarizes the results of an experiment that compares the relative overheads of gprof and csprof. We measured the execution times by running the benchmark in question five times and taking the median of the running times. The first column gives the name of the benchmark and the second column indicates its "base" execution time in seconds when run without profiling. The next two columns give statistics on the benchmark when run with instrumentation from gprof. The gprof and csprof overhead columns respectively report the percentage of the "base" execution time by which each profiler dilates execution. The column "gprof calls" reports a count of the number of procedure calls executed during the benchmark. Some benchmarks (gzip, vpr, gcc, eon, perlbench, vortex, and bzip) use multiple invocations of a program during their run. For such benchmarks, we summed procedure call counts across all invocations to provide the entry in the fourth column. Since the number of calls does not change from run to run, there is no need to provide the median.

As expected, gprof's overhead monitoring a program is directly related to the number of procedure calls the program makes. The programs vortex, eon, and parser all make several billion procedure calls and show drastic increases in their running times. Conversely, swim and lucas make a handful of calls and profiling overhead on their runs is barely noticeable.

equake, however, is an anomaly in these experiments. equake makes over a billion calls—a number comparable to eon—but equake's overhead is several orders of magnitude less than eon's overhead. Examining gprof's output, we found that three functions are responsible for 99.9% of the calls. These functions are short, one or two line functions and are called in the innermost loop of the application. In this case,
### Integer programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>base time (seconds)</th>
<th>gprof overhead %</th>
<th>gprof calls</th>
<th>csprof overhead %</th>
<th>csprof data file size (bytes)</th>
<th>csprof unsafe samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>479</td>
<td>53</td>
<td>1.960x\text{10}^9</td>
<td>2.1</td>
<td>270,194</td>
<td>3.2</td>
</tr>
<tr>
<td>175.vpr</td>
<td>399</td>
<td>53</td>
<td>1.558x\text{10}^9</td>
<td>2.8</td>
<td>98,678</td>
<td>1.0</td>
</tr>
<tr>
<td>176.gcc</td>
<td>250</td>
<td>78</td>
<td>9.751x\text{10}^8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>181.mcf</td>
<td>475</td>
<td>19</td>
<td>8.455x\text{10}^5</td>
<td>7.5</td>
<td>30,563</td>
<td>0.5</td>
</tr>
<tr>
<td>186.crafty</td>
<td>196</td>
<td>141</td>
<td>1.908x\text{10}^7</td>
<td>5.4</td>
<td>12,534,317</td>
<td>0.2</td>
</tr>
<tr>
<td>197.parser</td>
<td>700</td>
<td>167</td>
<td>7.009x\text{10}^9</td>
<td>3.7</td>
<td>12,083,741</td>
<td>22</td>
</tr>
<tr>
<td>252.eon</td>
<td>263</td>
<td>263</td>
<td>1.927x\text{10}^9</td>
<td>3.4</td>
<td>757,943</td>
<td>1.5</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>473</td>
<td>165</td>
<td>2.546x\text{10}^9</td>
<td>2.5</td>
<td>1,757,749</td>
<td>4.0</td>
</tr>
<tr>
<td>254.gap</td>
<td>369</td>
<td>39</td>
<td>9.980x\text{10}^8</td>
<td>4.2</td>
<td>2,215,955</td>
<td>0.8</td>
</tr>
<tr>
<td>255.vortex</td>
<td>423</td>
<td>230</td>
<td>6.707x\text{10}^9</td>
<td>3.9</td>
<td>6,060,039</td>
<td>7.0</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>373</td>
<td>112</td>
<td>3.205x\text{10}^8</td>
<td>3.8</td>
<td>180,790</td>
<td>0.2</td>
</tr>
<tr>
<td>300.twolf</td>
<td>568</td>
<td>59</td>
<td>2.098x\text{10}^8</td>
<td>2.6</td>
<td>122,898</td>
<td>0.1</td>
</tr>
</tbody>
</table>

### Floating-point programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>base time (seconds)</th>
<th>gprof overhead %</th>
<th>gprof calls</th>
<th>csprof overhead %</th>
<th>csprof data file size (bytes)</th>
<th>csprof unsafe samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>168.wupwise</td>
<td>351</td>
<td>85</td>
<td>2.233x\text{10}^9</td>
<td>3.2</td>
<td>559,178</td>
<td>0.02</td>
</tr>
<tr>
<td>171.swim</td>
<td>298</td>
<td>0.17</td>
<td>2,401</td>
<td>2.0</td>
<td>93,729</td>
<td>0.2</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>502</td>
<td>0.12</td>
<td>59,177</td>
<td>2.5</td>
<td>170,034</td>
<td>0.3</td>
</tr>
<tr>
<td>173.applu</td>
<td>331</td>
<td>0.21</td>
<td>219,172</td>
<td>3.9</td>
<td>317,650</td>
<td>0.1</td>
</tr>
<tr>
<td>177.mesa</td>
<td>272</td>
<td>67</td>
<td>1.658x\text{10}^9</td>
<td>4.0</td>
<td>56,676</td>
<td>2.7</td>
</tr>
<tr>
<td>178.galgel</td>
<td>251</td>
<td>5.5</td>
<td>1.490x\text{10}^7</td>
<td>2.8</td>
<td>756,155</td>
<td>0.03</td>
</tr>
<tr>
<td>179.art</td>
<td>196</td>
<td>2.1</td>
<td>1.110x\text{10}^7</td>
<td>1.5</td>
<td>76,804</td>
<td>0.03</td>
</tr>
<tr>
<td>183.equake</td>
<td>557</td>
<td>0.75</td>
<td>1.047x\text{10}^9</td>
<td>5.7</td>
<td>44,889</td>
<td>0.39</td>
</tr>
<tr>
<td>187.facercc</td>
<td>262</td>
<td>9.4</td>
<td>2.555x\text{10}^8</td>
<td>4.4</td>
<td>197,114</td>
<td>3.3</td>
</tr>
<tr>
<td>188.ammp</td>
<td>551</td>
<td>2.8</td>
<td>1.006x\text{10}^8</td>
<td>3.1</td>
<td>93,166</td>
<td>0.18</td>
</tr>
<tr>
<td>189.lucas</td>
<td>304</td>
<td>0.30</td>
<td>195</td>
<td>3.1</td>
<td>113,928</td>
<td>0.06</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>428</td>
<td>18</td>
<td>5.280x\text{10}^8</td>
<td>2.3</td>
<td>232,958</td>
<td>3.4</td>
</tr>
<tr>
<td>200.sixtrack</td>
<td>472</td>
<td>0.99</td>
<td>1.03x\text{10}^7</td>
<td>1.3</td>
<td>184,030</td>
<td>1.0</td>
</tr>
<tr>
<td>301.apsi</td>
<td>550</td>
<td>12</td>
<td>2.375x\text{10}^8</td>
<td>2.2</td>
<td>1,209,095</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 5.2: Results from the SPEC CPU2000 benchmark suite. The columns “gprof overhead” and “csprof overhead” show a percentage overhead relative to the “base time” column. A profiling overhead of 100% indicates execution took twice as long when profiling.
the compiler-added gprof instrumentation in these functions should result in a large amount of overhead during an equake run.

We found that the compiler was able to reduce the execution cost of the profiling code through optimization. These functions were so small that heuristics for inlining them would certainly decide to do so. Once the functions were inlined, the profiling code would be inlined as well. Since the profiling instrumentation was exposed in the intermediate representation of the compiler, the compiler determined that the profiling code was loop-invariant. After determining this, the profiling code was hoisted out of the loop, reducing the overhead of profiling during execution without compromising accuracy. Although we were unable to examine the inner workings of the compiler, we were able to compile an optimized binary which did not include the profiling code. We then used Tru64’s hiprof tool, which instruments object files with gprof-style instrumentation. Benchmarking this modified binary increased the runtime to 956 seconds, an overhead of 74%. This figure is comparable to the overhead observed for applications such as mesa or wupwise. Our experiences with equake indicate that the overhead of profiling instrumentation can be substantially reduced with an aggressive compiler.

The rightmost two columns of Table 5.2 show statistics pertinent to running the benchmark with csprof. The fifth column gives the overhead of profiling with csprof as a percentage of the “base” running time. The sixth column shows the size of the collected csprof data file; the same summing technique used for gprof’s call counts was also applied here. As the data file is very nearly a straight dump of csprof’s in-memory CCT, this column serves as an estimate for the added memory required by csprof. csprof’s memory usage is moderate on all programs, especially considering its high sample rate. If memory usage were to become a concern, modifications could be made to flush csprof’s profiling buffers when they exceeded a certain size. The third column in this group indicates the percentage of samples taken that were in
“unsafe” contexts (see section 3.3.1).

\texttt{csprof}'s overhead is dependent on several factors. Chief among these is the sample rate: more frequent samples imply more unwinds of the stack being undertaken. Even on programs that execute few calls and spend the majority of execution time in a few leaf procedures, unwinding can be expensive because of the extra processing that needs to be done on the first unwind step (see section 4.2.2). This factor is most noticeable on the floating-point benchmarks, which tend to execute many fewer calls than their integer brethren. However, even at the high sampling frequency used in this benchmark, \texttt{csprof}'s overhead is modest, generally no more than four or five percentage points.

Another factor is the call frequency of the program; programs with short procedures and deep call stacks will activate the trampoline more often, regardless of the sampling rate. Invoking the trampoline is expensive, as it saves about half the machine's register set to the stack. The cost of trampoline processing is most noticeable in \texttt{parser}, which makes many calls and incurs a large number of unsafe samples; these indicate frequent activations of the trampoline. Additionally, a program with deep call stacks and frequent returns causes the profiler to spend more time unwinding the stack and collecting nodes in the cached call stack.

The final factor is the breadth of the program's call tree. A wider call tree implies that nodes inside of the profiler's CCT contain lots of children. Early implementations of \texttt{csprof} used a linked list to store child nodes for nodes in the CCT. This scheme was found to perform poorly, as the majority of the profiler's time was spent searching for already-existing child nodes when inserting samples. This observation led to our recommendation in section 3.1.2 of storing the children in a balanced tree. We found using a balanced tree instead of a linked list to store child nodes led to a significant reduction in \texttt{csprof}'s overhead (a percentage point on two on “inner-loop” floating point codes such as \texttt{swim} and \texttt{sixtrack}), although searching the child nodes is still a significant factor in \texttt{csprof}'s overhead. The cost of this search to find where to
record a sample is the single most important determinant of cost once the sampling frequency is chosen.

Unlike *gprof*, *csprof* does not drastically inflate execution time when profiling an application with a large number of procedure calls. Instead, *csprof* produces consistent overhead across a wide variety of codes. Even when the number of calls is low, *csprof*’s overhead is comparable to *gprof*’s. *csprof* performs comparably to *gprof* in the average case while performing significantly better in call-intensive applications.

### 5.3 Accuracy comparison

Simply comparing overheads, however, does not provide an accurate measure of whether one profiler is to be preferred over another. A comparison between the information provided by two profilers should be made as well. If the profiler with lower overhead does not provide the information needed, or communicates inaccurate information, then the lower overhead advantage is meaningless. In this section, we show that the samples taken by *csprof* accurately portray the time spent in each of the application’s procedures. We do this by comparing flat profiles generated by *csprof* to flat profiles gathered via DCPI [3], a low-overhead system-wide sampling profiler. Using DCPI enables us to obtain data which closely reflects the actual behavior of the application while not distorting the behavior of the application by adding timing routines. For the sake of completeness, we also compare the flat profiles collected by *gprof* to DCPI.

Accuracy comparisons should concern themselves with two measures of accuracy: how much the cost of individual functions is distorted and how much the cost of individual functions is dilated. To compute both of these measurements, we require the time spent in each function in the program; this data is obtained from DCPI. We measure the total amount of distortion for the functions in program $p$ with the following equation:
\[ \sum_{f \in \text{functions}(p)} |P_x(f) - P_{depi}(f)| \quad (5.1) \]

\(P_x(f)\) returns the percentage of time consumed by \(f\) when \(p\) is profiled by \(x\). Ideally, this summation should be zero, indicating that the profiler reports the same fraction of time for each function as a low-overhead flat profiler. We measure the distortion (which includes dilation) for the functions in program \(p\) with the following equation:

\[ \frac{\sum_{f \in \text{functions}(p)} |T_x(f) - T_{depi}(f)|}{\text{execution.time}(p)} \quad (5.2) \]

\(T_x(f)\) returns the time consumed by \(f\) when \(p\) is profiled by \(x\). This summation should be close to the actual overhead of profiling. As this number moves away from the actual overhead of profiling, it indicates that the profiler underreports and/or overreports the time spent in individual functions.

We do not need to evaluate the accuracy of \texttt{csprof}'s sampled edge counts as they are exact. \texttt{gprof} collects edge count data (at great cost, as seen in section 5.2) to approximate the distribution of samples to call contexts. As discussed in section 2.2, this distribution can be inaccurate for particular types of calling patterns. In contrast, \texttt{csprof}'s edge counts are precisely attributed to their calling contexts, dispensing with the need for heuristics. \texttt{csprof}'s sampled edge counts serve to record exactly how many unique calls are represented by the samples taken rather than serving as an estimator of what the samples mean.

Table 5.3 shows the results of our accuracy evaluation. The second and third columns show the results of computing equation 5.1 for \texttt{csprof} and \texttt{gprof}, respectively. On nearly every integer benchmark, \texttt{csprof} gives a more accurate profile than \texttt{gprof}. The difference between them is especially significant for \texttt{eon}, which is written in C++ and is representative of modern modular codes. The notable exception in
### Integer benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>csprof distortion</th>
<th>gprof distortion</th>
<th>csprof time dilation</th>
<th>gprof time dilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.zip</td>
<td>1.3</td>
<td>6.9</td>
<td>2.0</td>
<td>52</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1.9</td>
<td>7.6</td>
<td>4.3</td>
<td>52</td>
</tr>
<tr>
<td>181.mcf</td>
<td>51</td>
<td>30</td>
<td>7.4</td>
<td>19</td>
</tr>
<tr>
<td>196.crafty</td>
<td>0.98</td>
<td>15</td>
<td>5.0</td>
<td>132</td>
</tr>
<tr>
<td>197.parser</td>
<td>6.9</td>
<td>15</td>
<td>3.6</td>
<td>163</td>
</tr>
<tr>
<td>252.eon</td>
<td>1.7</td>
<td>120</td>
<td>3.0</td>
<td>243</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>5.2</td>
<td>9.0</td>
<td>3.0</td>
<td>162</td>
</tr>
<tr>
<td>254.gap</td>
<td>9.3</td>
<td>7.8</td>
<td>4.1</td>
<td>38</td>
</tr>
<tr>
<td>255.vortex</td>
<td>2.9</td>
<td>15</td>
<td>3.7</td>
<td>221</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>5.8</td>
<td>16</td>
<td>3.7</td>
<td>110</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0.74</td>
<td>6.9</td>
<td>3.1</td>
<td>57</td>
</tr>
</tbody>
</table>

### Floating-point benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>csprof distortion</th>
<th>gprof distortion</th>
<th>csprof time dilation</th>
<th>gprof time dilation</th>
</tr>
</thead>
<tbody>
<tr>
<td>168.wupwise</td>
<td>9.4</td>
<td>8.1</td>
<td>3.1</td>
<td>83</td>
</tr>
<tr>
<td>171.swim</td>
<td>1.2</td>
<td>0.44</td>
<td>2.1</td>
<td>0.15</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>0.85</td>
<td>0.44</td>
<td>2.4</td>
<td>0.09</td>
</tr>
<tr>
<td>173.aplud</td>
<td>0.95</td>
<td>0.67</td>
<td>4.0</td>
<td>0.24</td>
</tr>
<tr>
<td>177.mesa</td>
<td>5.2</td>
<td>15</td>
<td>3.9</td>
<td>66</td>
</tr>
<tr>
<td>178.galgel</td>
<td>2.1</td>
<td>3.2</td>
<td>2.6</td>
<td>5.2</td>
</tr>
<tr>
<td>179.art</td>
<td>0.37</td>
<td>1.0</td>
<td>1.5</td>
<td>2.1</td>
</tr>
<tr>
<td>183.equate</td>
<td>3.0</td>
<td>4.4</td>
<td>5.5</td>
<td>0.70</td>
</tr>
<tr>
<td>187.facerec</td>
<td>4.4</td>
<td>1.5</td>
<td>4.3</td>
<td>9.2</td>
</tr>
<tr>
<td>188.ammp</td>
<td>4.1</td>
<td>3.2</td>
<td>3.0</td>
<td>2.7</td>
</tr>
<tr>
<td>189.lucas</td>
<td>6.4</td>
<td>0.12</td>
<td>3.0</td>
<td>0.26</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>18</td>
<td>7.5</td>
<td>2.4</td>
<td>18</td>
</tr>
<tr>
<td>200.sixtrack</td>
<td>0.44</td>
<td>0.33</td>
<td>1.3</td>
<td>0.96</td>
</tr>
<tr>
<td>301.apsi</td>
<td>13</td>
<td>7.1</td>
<td>2.2</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.3: Distortion and dilation caused by csprof and gprof relative to "base" measurements obtained from DCPI. The "distortion" columns are computed using equation 5.1 for each profiler; the "dilation" columns are computed using equation 5.2. csprof used a sampling rate of 1000 samples/second; gprof used a sampling rate of 1000 samples/second.
the integer benchmarks is mcf, for which csprof drastically distorts the individual function percentages. We will examine this case shortly.

Floating-point benchmarks, however, show a slightly different story. While csprof is more accurate on a few benchmarks, such as quake, art, galgel, and mesa, gprof proves more accurate for apsi and fma3d. On other benchmarks there is little difference between the two profilers.

A similar story occurs when considering the amount of distortion; the fourth and fifth columns of 5.3 show the results of computing equation 5.2 for csprof and gprof, respectively. As expected, csprof makes a strong showing on the integer benchmarks with this metric. This result is not surprising since, as noted above, equation 5.2 is closely related to the amount of overhead a profiler incurs and csprof incurred low overhead for the integer benchmarks. A similar situation occurs on the floating-point benchmarks with this second metric: csprof's overhead is lower than that of gprof on a majority of the floating-point benchmarks and it wins in similar fashion here.

5.3.1 Reducing the sample rate

Earlier we noted a unusual measurement for csprof: its accuracy on the benchmark mcf was much worse than its accuracy on every other benchmark. We were puzzled by this finding and turned to DCPI to assist in diagnosing the problem. We ran the same mcf binary used to record the “base” numbers in Table 5.2, using DCPI to gather processor-specific information such as the number of cache misses incurred. We then used DCPI to profile csprof profiling mcf to see what effect csprof was having on mcf’s behavior.

Examining the DCPI data and the profile gathered by csprof turned up several irregularities. First, we found that a csprof-enabled run incurred instruction TLB misses, whereas a “bare” run did not. We attribute this to the sampling csprof performs, which necessitates executing instructions in a different area of the virtual address space than the application code. This should happen in nearly any profiled
program; however, mcf is smaller than most codes (the mcf binary is approximately
a quarter of the size of csprof’s shared library). Second, we found that csprof was
drastically underestimating one of the key procedures in mcf, refresh_potential,
which traverses a network of interconnected nodes. DCPI indicated that this function
was responsible for nearly 40% of the data TLB misses and all of the load/store
replay traps, a consequence of the many memory locations it touches during its work.
Third, we noticed that csprof always attributed the exact same number of samples
to refresh_potential and that this number of samples was the exact number of
calls as measured by gprof. This was extremely peculiar, as no other function in
the benchmark suite exhibited this behavior. Furthermore, DCPI indicated that the
percentage of time being spent in refresh_potential was virtually identical on the
profiled and unprofiled runs. Therefore, csprof was not unduly skewing the behavior
of the program, just failing to attribute information to the correct place.

Based on the above observations, we conjecture that the high number of TLB
misses creates dead zones in refresh_potential where the expiration of the interval
timer csprof uses is not processed until the end of the zone. Such dead zones have
the effect of moving samples that would have fallen in a dead zone to the end of the
dead zone. Furthermore, if the timer expires in a dead zone, then the timer will not
be reset until the end of the dead zone. If this occurs often, it will cause the timer
to fire less frequently than expected. Indeed, this seems consistent with the behavior
that we observed above.

Given the previous conjecture, We hypothesized that reducing the sample rate
might improve csprof’s accuracy on mcf, as fewer samples would fall in dead zones,
leading to fewer misattributed samples. To test this hypothesis, we ran a second set
of experiments, using the same settings as described in section 5.1, except that we
only evaluated csprof and we used a lower sampling rate of 100 samples/second.
Table 5.4 shows the results of evaluating equation 5.1 using the data gathered by this
second set of experiments for all benchmarks. We provide the data for gprof for the sake of comparison.

In nearly all cases, the accuracy at 100 samples/second is at least as good as the accuracy at 1000 samples/second. We see that mcf’s accuracy improves dramatically at lower sampling rates. The lone outlier in this set of experiments is gzip. While csprofi’s accuracy is still comparable to gprof for gzip, the accuracy at the lower sampling rate is noticeably worse than at the higher sampling rate. We attribute this to the gzip benchmark being composed of five runs of the gzip program on five different sets of input data. The average time for each execution, then, is about ninety-six seconds, with some executions taking less and some taking more. At 100 samples/second, then, the average profile run is only able to collect several thousand samples, which is clearly not enough to form an accurate picture of gzip. Examining the profiles at 100 samples/second showed that the profiles tended to overestimate the percentage of time spent in longest.match, the core of gzip’s compression algorithm. None of the other benchmarks had such short execution times and thus csprofi was able to obtain a more accurate picture of their behavior.

5.4 Benefits of extra information

One of the motivating ideas behind csprofi is the collection of full calling context, in contrast to gprof’s recording only the immediate caller. We show in this section that the extra information gathered by csprofi can expose behavior that a gprof profile is unable to detect.

An example of the behavior in which we are interested occurs in the SPEC benchmark wupwise. wupwise is a numerical simulation in lattice gauge theory that computes quark propagations within a chromodynamic background gauge field. zaxpy, which multiples a constant times a vector and then adds another vector, is one of the key routines in wupwise; it is called from several different places in the code and accounts for 18% of the execution time. 77% of the calls to zaxpy come from gammul,
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>cprof distortion</th>
<th>gprof distortion</th>
</tr>
</thead>
<tbody>
<tr>
<td>164.gzip</td>
<td>9.4</td>
<td>6.9</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1.5</td>
<td>7.6</td>
</tr>
<tr>
<td>181.mcf</td>
<td>7.6</td>
<td>30</td>
</tr>
<tr>
<td>196.crafty</td>
<td>1.3</td>
<td>15</td>
</tr>
<tr>
<td>197.parser</td>
<td>2.4</td>
<td>15</td>
</tr>
<tr>
<td>252.eon</td>
<td>2.5</td>
<td>120</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>5.2</td>
<td>9.0</td>
</tr>
<tr>
<td>254.gap</td>
<td>8.7</td>
<td>7.8</td>
</tr>
<tr>
<td>255.vortex</td>
<td>3.3</td>
<td>15</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4.5</td>
<td>16</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0.87</td>
<td>6.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>cprof distortion</th>
<th>gprof distortion</th>
</tr>
</thead>
<tbody>
<tr>
<td>168.wupwise</td>
<td>5.1</td>
<td>8.1</td>
</tr>
<tr>
<td>171.swim</td>
<td>1.2</td>
<td>0.44</td>
</tr>
<tr>
<td>172.mgrid</td>
<td>1.0</td>
<td>0.44</td>
</tr>
<tr>
<td>173.applu</td>
<td>0.89</td>
<td>0.67</td>
</tr>
<tr>
<td>177.mesa</td>
<td>4.6</td>
<td>14</td>
</tr>
<tr>
<td>178.galgel</td>
<td>6.0</td>
<td>3.2</td>
</tr>
<tr>
<td>179.art</td>
<td>0.23</td>
<td>1.0</td>
</tr>
<tr>
<td>183.equake</td>
<td>3.1</td>
<td>4.4</td>
</tr>
<tr>
<td>187.facerc</td>
<td>4.2</td>
<td>1.5</td>
</tr>
<tr>
<td>188.ammp</td>
<td>3.6</td>
<td>3.2</td>
</tr>
<tr>
<td>189.lucas</td>
<td>5.9</td>
<td>0.12</td>
</tr>
<tr>
<td>191.fma3d</td>
<td>14</td>
<td>7.5</td>
</tr>
<tr>
<td>200.sixrtrack</td>
<td>0.70</td>
<td>0.33</td>
</tr>
<tr>
<td>301.apsi</td>
<td>11</td>
<td>7.1</td>
</tr>
</tbody>
</table>

Table 5.4: Distortion (equation 5.1) caused by cprof and gprof relative to “base” measurements obtained from DCPI. cprof used a sampling rate of 100 samples/second; gprof used a sampling rate of 1000 samples/second.
which in turn is called from only two places, muldeo and muldoe. This is the extent of the information gprof is able to provide, as gprof only records only level of calling context. gprof is unable to indicate if particular calling contexts of zaxpy are more expensive than others.

Figure 5.1 is a portion of wupwise's call graph, put together from information collected by csprof. gammul is called from two different call sites in both muldeo and muldoe. In each case, when gammul is called from one of the call sites, it makes an order of magnitude more calls to zaxpy than when it is called from the other
call site. We also note that 

\texttt{cprof} has confirmed \texttt{gprof}'s average time assumption, as each sample taken in \texttt{zaxpy} resulted in a unique return. \texttt{gprof} collects enough information to distinguish between \texttt{gamma}\texttt{\_}\texttt{mul}'s calls to \texttt{zaxpy}.\footnote{Commonly available analysis tools do not expose this level of detail to the end user, instead aggregating all calls of \texttt{zaxpy} from \texttt{gamma}\texttt{\_}\texttt{mul} into a single number.} But \texttt{gprof} does not provide enough information to see how the calling patterns to \texttt{zaxpy} differ based on the path taken prior to calling \texttt{zaxpy}. \texttt{cprof} provides the information that \texttt{gprof} is unable to provide and often provides more informative profiles at lower cost than \texttt{gprof}.

\section{Summary}

In summary, then, \texttt{cprof} incurs lower overhead than \texttt{gprof} on 18 out of the 25 SPEC benchmarks, particularly call-intensive programs. While \texttt{cprof} does not always incur lower overhead than \texttt{gprof}, it is important to remember that \texttt{cprof} enables accurate attribution of costs to call paths by collecting more information that \texttt{gprof}. Even on non-call-intensive codes, \texttt{cprof}'s overheads are still reasonable, averaging 3.3\%. On call-intensive codes, such as \texttt{eon} and \texttt{parser}, \texttt{cprof} delivers its more accurate results with significantly lower overhead than \texttt{gprof}. For \texttt{eon} in particular, \texttt{cprof}'s monitoring overhead is 3.4\% with a relative distortion of 2.5\%, whereas \texttt{gprof}'s monitoring over is 263\% with a relative distortion of 120\%. Furthermore, for those few codes where \texttt{gprof} turns in more accurate results than \texttt{cprof}, the difference in distortion is slight. Even on these codes, using \texttt{cprof} would be an acceptable choice, especially when \texttt{cprof}'s ability to provide full calling context for samples in conjunction with its ability to profile unmodified, optimized binaries without prior preparation is considered.
Chapter 6

Conclusions and future work

In the previous chapters, we presented the design of a portable, sampling-based, context-sensitive profiler. In this chapter, we summarize our design and present some possible applications for this technology. We also describe some directions for future work.

6.1 Profiler Design Analysis

The portability of our profiler stems from its usage of procedure descriptors to determine the behavior of the application code at runtime. While procedure descriptors are often compiler generated (indeed, they are required by the Itanium ABI [11] and the PowerPC ABI [18]), there is no reason that procedure descriptors cannot be synthesized by binary analyzers after the program has been compiled. Using procedure descriptors enables the profiler to run well on highly optimized code, which enables the performance analyst to study the actual application in question, rather than a recompiled version with less optimization or instrumentation.

Previous approaches to context-sensitive profiling operate by inserting instrumentation to track the context of the program at runtime. While this technique is flexible in the type of profiling it can accommodate, its overhead can be unacceptable for call-intensive programs and its accuracy suffers on codes with frequent calls to small functions, as our experiments in section 5.2 showed. Sampling provides for controllable overhead; walking the stack at every sample event to determine the active procedures provides the context of each event. We showed how to use a trampoline as a sentinel, which improves the efficiency of stack unwinding.
While the use of a trampoline has been explored prior to this thesis, previous approaches have not considered highly optimized code where there are not always stack frames for procedures or frame pointers for procedure activations. Restricting the compiler from performing these optimizations unnecessarily limits the performance of the application. Using procedure descriptors enables the profiler to cope with optimized frames and calling sequences. This thesis also showed that a trampoline can be used to attribute return counts to edges in the collected call graph as well as to improve the efficiency of stack sampling. These return counts correlate strongly with total call counts collected via instrumentation, but come at a far lower runtime cost.

We evaluated an implementation of this design for the Tru64/Alpha platform, csprof, across a wide variety of application programs. Our results showed that csprof provided lower average-case and worst-case overhead than the standard gprof call graph profiler. Furthermore, the accuracy of csprof’s profiles was higher when compared to those of gprof, as csprof perturbed the application’s behavior less.

### 6.2 Applications

The profiler described in the previous chapters was written with large-scale scientific codes as its primary target. However, it is useful for profiling many other kinds of applications as well. Desktop programs, for example, with complex interactions between libraries (e.g. graphical user interface toolkits) and application code, could benefit from identifying why time is being spent in the libraries on behalf of the application proper. Conversely, toolkit developers could use this profiler to discover which parts of the toolkit do not scale for certain applications. New scalable interfaces could be developed or the old interfaces improved.

As the profiler’s overhead is relatively low, the profiler could be profitably used for online profiling of applications with an eye towards adaptive optimization. For instance, in Java, once a method has been compiled to native code inside a just-in-time runtime, the method (and its calling context) could be monitored by the call
stack profiling described herein. The context trees gathered from this monitoring could then be used to tune the compiled code by noticing which paths were most likely and specializing for those cases. While the profiler is not suited to basic-block profiling [4, 6], determining the calling context of sampled functions makes it ideally suited for profiling object-oriented applications. In these applications, identifying profitable contexts in which to inline (avoiding the overhead of method dispatch) and determining the most likely calling object (fast-tracking the method dispatch process) can be very profitable optimizations.

Using this profiler design with a runtime system like a Java Virtual Machine raises several issues. If the runtime system allows code fragments to move (e.g. during garbage collection), then each garbage collection effectively starts a new epoch. Epochs in such runtime systems are also heavyweight objects, needing to record information about every compiled function, whereas epochs in your system need only record the name of a library and the offset at which its text section begins. While these problems can be overcome, a more fine-grained epoch abstraction will be necessary. Adaptive runtime systems also present challenges in mapping executable code back to source code, since the executable code is no longer available after the system has finished executing the program.

6.3 Future work

A profiler design that captures call path information and does so at low cost has many applications. In this section, we offer ideas for extensions of the profiler that enable its benefits to be realized in more areas, its profiles to be more effective for analysis of performance problems, and further performance improvements to the profiler itself.
6.3.1 Extensions

Visualization of call path information

Collecting performance metrics along with their associated calling contexts is only one step in effective performance analysis. Collected data must also be presented in a manner that assists in identifying performance bottlenecks.

Given that the program is conveniently thought of as a graph, an obvious idea is to draw the collected profile data as a graph, with visual indications of which functions (nodes) accumulated greater numbers of samples. However, our experience is that for even moderately-sized codes, the graphs are generally quite large and unwieldy. We have experimented with providing mechanisms to enable the user to interactively explore the call graph: a user requests that certain nodes be expanded or collapsed to display or hide their callees as appropriate. We found that this mechanism exposes a limitation of many graph layout algorithms: the graph is restructured after expanding or collapsing nodes, which proves confusing to the user. Another method, proposed by Bernat and Miller [7], is to draw rectangles representing functions in the profiled program; caller-callee relationships are then shown by nesting the rectangles. The relative sizes of rectangles indicates the number of samples recorded in the functions they represent.

In the above discussion, we proposed the idea of enabling the user to interactively explore the collected call tree. Hall [20] described a system that does exactly this, promoting the idea of “call path refinements.” Call path refinements are filters specifying which call paths should be analyzed (displayed, etc.): “show me profiling data for all paths that call the differential equation solver and that also perform mutex operations.” This functionality permits the user to quickly focus on “hot” paths via iterative refinement of filter criteria. Another benefit of Hall’s approach, as shown in the work by Ammons et al on BOTTLENECKS [2] is that call path refinements enable aggregation across call paths, which may uncover bottlenecks that would not
be apparent using the graph-drawing approach above. Call path refinements combine the best features of flat profiles (easily identify “hot spots”) with the benefits of call path information (where was I during those “hot spots”?).

Yet another possibility is to modify a visualization tool for flat profiles, such as HPCView, part of HPCToolkit [26], to support call path profiles. HPCView works by correlating profile information with source code and aggregating the data hierarchically according to the static structure of the program. This approach encourages a top-down analysis procedure, which enables the user to quickly identify the hot spots of the program by adjusting the scope of the static structure displayed. One difficulty in adapting a flat profile viewer to visualize call path profiles is that a call path profile naturally splits functions across their contexts; an adaptation must implement a good way of enabling easy and intuitive switches between “global” views and context-sensitive views.

Profiling multi-process applications

Large-scale scientific codes often run on a cluster of machines using a communication library such as MPI to pass messages between nodes. Profiling such applications introduces several of the problems discussed in Chapter 1. For example, a profile may indicate that a large amount of time is being spent passing messages between cluster nodes. However, the message passing itself is already well-optimized; the problem lies in the routines communicating excessively. Our profiler seems like an ideal match to profiling these sorts of applications.

Unfortunately, profiling MPI codes with our profiler introduces an issue of its own: the asynchronous nature of the sample interrupts disrupts the program to a far greater extent than the overhead measured in Chapter 5. To see how this happens, consider the following example, with three processes $A$, $B$, and $C$. Assume that process $A$ is computing and then receives a sample interrupt. The sample interrupt must be serviced; meanwhile, process $B$ is expecting data from $A$ and is forced to wait. Even
after $A$ resumes and sends data to $B$, $C$ may now be forced to wait because $B$ is “behind” in doing its work. Meanwhile, other sample interrupts are occurring on $A$, $B$, and $C$, causing further delays. The delay introduced by one sample event can be magnified far beyond the delay caused by its servicing.

One solution to this problem would be to utilize the “heartbeat” monitoring function of many clustering software packages. The nodes of the cluster send a “heartbeat” to the central server (usually the batch scheduling server) indicating that they are up and ready to accept job submissions. In addition to notifying the batch scheduler that the node can accept jobs, the heartbeat could also be used as a sample source for the currently running application. Assuming that all of the nodes are roughly synchronized in sending heartbeat signals, subsuming the heartbeat monitor provides a sample source that will not cause the cascading delays described in the previous paragraph. One problem is that the heartbeat may not occur often enough to provide enough samples for short-running jobs.

6.3.2 Implementation improvements

Ports to other architectures

We have argued throughout this thesis that the techniques used in our call path profiler design are portable across platforms. To date, we have only demonstrated their utility on one particular platform. This fact suggests limits in the applicability of our technique, especially since the necessary interface for procedure descriptors was inspired by the platform on which our implementation runs.

To demonstrate the feasibility of our approach on other platforms, we have constructed a prototype for the Linux/x86-64 platform is in its early stages; at present, this version is able to cope with programs that are compiled to use a frame pointer. Due to the CISC instructions of the x86-64 architecture used to call and return from procedures, procedure descriptors unnecessary and one can unwind with a very simple algorithm. The ABI for Linux/x86-64, however, does not mandate the user of frame
pointers and our current prototype is thus unsuitable for optimized code.

We are in the process of extending the x86-64 version to handle programs that do not use frame pointers using libunwind [28], a portable library for call stack unwinding. libunwind provides a call stack unwinding abstraction over the DWARF2 [22] procedure descriptors required by the Linux/x86-64 ABI. Like the Tru64/Alpha unwinding library, libunwind relies on accurate procedure descriptors to be generated by the compiler. At this time, the de facto compiler for the Linux/x86-64 platform, GCC, does not generate procedure descriptors for procedure epilogues. While procedure descriptors for epilogues are not necessary for exception handling purpose, they are necessary for the asynchronous, timer-based profiling described in section 4.1, as a sample interrupt may occur in the epilogue of a procedure, necessitating unwinding from that point. Work on this platform, then, will necessarily involve modifying the compiler as well as modifying the profiler.

Using sample bits instead of a trampoline

Using Whaley’s sample bit approach, which marks each call frame active when a sample is taken by setting the low-order bits in the frame’s return address, appears preferable to our trampoline-based approach. Whaley’s approach only requires work at each sample event, while the trampoline approach incurs overhead at procedure returns in addition to sample collection time. We could not use Whaley’s sample bit approach on Tru64/Alpha because the Alpha processor does not automatically mask off the low-order bits of a return address on procedure return. However, modifying an application binary and its associated libraries to include a mask instruction prior to every return instruction would provide the same effect in software. This modified binary could then be profiled by a sample-bit aware version of csprof and should provide lower overhead than our trampoline-based approach. Unfortunately, most binary-rewriting tools only support the adding function calls at selected places in the code, rather than adding arbitrary instructions in a procedure epilogue. Creating a
binary rewriting tool capable of inserting arbitrary code at specified points in the code would enable even more efficient call path profiling than the implementation demonstrated in this thesis.
Bibliography


