Low-Overhead Call Path Profiling of Unmodified, Optimized Code

Nathan Froyd
froydnj@cs.rice.edu

John Mellor-Crummey
johnmc@cs.rice.edu

Rob Fowler
rjf@cs.rice.edu

Rice University, Houston, TX

ABSTRACT

Call path profiling associates resource consumption with the calling context in which resources were consumed. We describe the design and implementation of a low-overhead call path profiler based on stack sampling. The profiler uses a novel sample-driven strategy for collecting frequency counts for call graph edges without instrumenting every procedure's code to count them. The data structures and algorithms used are efficient enough to construct the complete calling context tree exposed during sampling. The profiler leverages information recorded by compilers for debugging or exception handling to record call path profiles even for highly-optimized code. We describe an implementation for the Tru64/Alpha platform. Experiments profiling the SPEC CPU2000 benchmark suite demonstrate the low (2%-7%) overhead of this profiler. A comparison with instrumentation-based profilers, such as `gprof`, shows that for call-intensive programs, our sampling-based strategy for call path profiling has over an order of magnitude lower overhead.

1. INTRODUCTION

In modern, modular programs, it is important to attribute the costs incurred by each procedure to the different contexts in which each procedure is called. The costs of communication primitives, operations on data structures, and library routines can vary widely depending upon their calling context. Because there are often layered implementations within applications and libraries, it is insufficient to insert instrumentation at any one level, nor is it sufficient to distinguish costs based only upon the immediate caller. For example, the cost of a call to `memcpy`—a standard library routine that copies data—depends on the size of the data region on which it operates. `memcpy` is called from many places in the MPICH implementation of the MPI [13] communication library. In addition, many applications encapsulate MPI calls within a data-exchange layer that is in turn called from multiple places. To understand the performance implications of data copying in such applications, it can thus be necessary to attribute costs at all of these levels. To do this in general requires an efficient data collection mechanism that tracks costs along entire call paths. Since understanding and improving the on-node performance of MPI programs is a key factor in overall performance, context-sensitive profiling is of particular importance for large-scale parallel systems.

An ideal profiler should have the following properties. First, it should collect performance measurements with low, controllable overhead; this makes it feasible to collect profile information during production runs as well as during development. Second, it should identify differences in the cost of calls according to calling context; this is needed to enable accurate performance analysis in the presence of layered implementations. Third, the profiler should not require changes to the application build process; for large applications, changing the build process can be daunting and recompilation can be time consuming. Fourth, it should support accurate profiling at the highest level of compiler optimization; this enables profiling of production code as well as development code. Fifth, it should also attribute costs to library functions, even if a library is available only in binary form. Finally, it should minimize distortion of the performance of the application under study; otherwise, any performance measurements gathered may not be representative.

Two general approaches to keeping track of calling contexts in profilers are described in the literature. *Exhaustive profilers* [12, 9, 20, 22] maintain calling context information at every function call and sometimes at function returns as well. This is done by adding execution-time instrumentation at call sites or in procedure bodies to record nodes and edges in graph or tree. The execution costs in such profilers can be measured either by inserting timing calipers around procedure bodies, or through sampling. In contrast, *statistical profilers* [6, 5, 7, 15, 23] both measure costs and attribute them to calling contexts through periodic sampling of the stack of active procedures without any instrumentation of procedure bodies.

The most widely-used call graph profiler, `gprof` [12], relies on instrumentation, inserted by a compiler or through binary rewriting, to record the number of times each procedure is called from each unique call site and the location of each call site. `gprof` relies on statistical sampling of the program counter to estimate time spent in each procedure. To attribute execution time to call sites, `gprof` uses call site counts to apportion a procedure's time among the sites that

ICS'05 Cambridge, Massachusetts

```
#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: A short program that illustrates the shortcomings of instrumentation-based context tracking.

Profiler	Platform	Dilation factor
gprof	Linux/Pentium4 2.0GHz	3
gprof	OS X/PowerPC 7450 1.33 GHz	4
hiprof	Tru64/Alpha 667MHz	14
VTune	WinXP/Pentium4 1.8GHz	31

Table 1: Results from running the program in Figure 1 on various platforms using instrumentation-based profiling. A dilation factor of 2 means an instrumented binary took twice as long to run as an uninstrumented binary.

call it. By attributing costs in this fashion, `gprof` makes a key simplifying assumption: that the amount of time spent in a procedure call is context independent. This assumption is often invalid [18, 21] and makes it impossible to attribute context-dependent costs for routines like `memcpy` or MPI communication calls. Furthermore, `gprof`'s reliance on instrumentation within procedure bodies substantially dilates the execution time of programs with high call frequencies and the overhead of instrumentation code significantly distorts the cost of small functions, which are common in object-oriented programs.

The program in Figure 1 illustrates these shortcomings of instrumentation-based context tracking. The execution time for function `c` depends on its argument, which makes its execution time dependent on its calling context. Function `d` is small. `gprof` adds instrumentation to d to count calls and the immediate calling context; this significantly inflates the fraction of time that `gprof` attributes to `d`. Calls to `c` (through `*f`) from `a` take twice as long as calls to `c` (through `*f`) from `b`, but there are only half as many, so the costs should be approximately equal. Instead, `gprof` attributes 2/3 of the time spent in `c` to `b` and 1/3 to `a` when the time spent in `c`. Table 1 shows that `gprof`-style instrumentation used by `gprof` and Compaq's `hiprof` dilates execution time from a factor of three to a factor of 14 when compared to an uninstrumented executable. While Intel's `VTune` divides the time of `c` equally (within 5%) between `a` and `b`, on a Pentium 4/Windows platform it dilates the execution time of by a dramatic factor of 31![1] In addition to dilating the overall time of the instrumented run, VTune reports an execution time for the example itself that is a factor of 2.4 times the uninstrumented time. While this example is extreme, it illustrates the impact of instrumentation-based tracking on the dilation of execution time in call-intensive programs such as object-oriented programs.

[1]VTune documentation admits that the cost of call-path profiling can be big, so it recommends sparing use on parts of an application rather than whole applications [16].

Apple's Shark [4] is a statistical call-path profiler based on stack sampling. Shark correctly apportions the execution cost of `c` between `a` and `b`; however, it contains no support for counting calls from each context, so it does not have a mechanism for estimating the cost per call or cost per call by context. Therefore, Shark cannot tell us that the calls to `c` from `b` are half the cost of calls from `a`. This is precisely what a call graph profiler needs to do: identify differences in the cost of calls according to context. Shark solves half the problem.

The rest of the paper describes `csprof`, an efficient call path profiler with the desired profiler properties listed earlier. `csprof` measures performance with low, controllable overhead by periodically sampling the call stack of active procedures. As we explain in the next section, `csprof` uses a novel and efficient technique to associate counts with call graph edges; this enables it to estimate the average cost of a call in each unique context. `csprof` profiles unmodified application binaries that have been compiled at the highest levels of optimization; its only requirement is that executables contain the minimal debugging information needed to unwind the call stack during execution. Since `csprof` does not depend on compiler-inserted instrumentation, it can properly attribute costs to library routines available only in binary form. In addition to accurately attributing costs to full call chains, `csprof`'s sampling-based strategy for collecting call graph profiles typically has overhead of only a few percent.

In the next section, we describe `csprof`'s high level design. In Section 3, we add detail for and implementation for the Tru64/Alpha platform. In Section 4, we compare the overhead of `csprof` to that of `gprof` a representative instrumentation based profiler. We outline the future directions of this work in in Section 5 and summarize our conclusions in Section 6.

2. HIGH-LEVEL DESIGN

We first briefly describe how `csprof` initiates profiling of unmodified application binaries. The remainder of this section provides a detailed description of our sampling-based strategy for collecting call path profiles and how it enables us to accurately attribute costs to procedure calls in context. We conclude the section by describing the complexities of dealing with large, real-world applications.

To initiate profiling of an unmodified, dynamically linked[2] application binary, we instruct the loader to pre-load our profiling library.[3] This library's initialization routine creates the data structures for the profiler's state and initiates profiling. The application is then loaded and begins to execute. After the application exits, the profiling library finalization routine halts profiling and writes the collected profile data to a file.

2.1 Sample Events

There are two modes for the generation of sample events. *Asynchronous events* are events that, from the point of the program, are not generated by direct program activity. To monitor a metric that occurs asynchronously, there are three

[2]Handling statically linked binaries is simple in principle but outside the scope of this paper.

[3]On ELF systems, this is done by setting the `LD_PRELOAD` environment variable.

```
void *malloc(size_t bytes) {
  prof_state_t *state = get_prof_state();
  state->bytes_alloc += bytes;

  if(state->bytes_alloc >= T) {
    /* attribute multiple samples if necessary */
    unsigned int count = state->bytes_alloc / T;
    state->bytes_alloc = state->bytes_alloc % T;

    record_sample(state, get_caller_context(), count);
  }

  return system_malloc(bytes);
}
```

Figure 2: A replacement for `malloc` that generates profile events for the number of bytes allocated. `T` is the number of bytes allocated before a sample event occurs.

requirements: the system can generate a trap at the end of an interval in which a known amount of a resource has been consumed; the operating system can deliver context information about the machine state to the profiler; and that the instruction pointer in the delivered context is reasonably close to an instruction that "consumes" the resource. A POSIX signal delivers a *context* structure: a structure containing the values of hardware registers at the point of the trap. Typical sources of asynchronous events are `SIGPROF` signals triggered by a system interval timer, or overflow traps generated by hardware performance counters. When an event occurs, the "cost" associated with the event is charged to the target of the instruction pointer, and, implicitly, to the entire current chain of call sites.

A second instrumentation mode is the monitoring of *synchronous events*: those events triggered by direct program action. Synchronous events of interest include thresholds related to the amount of memory that has been allocated or the number of bytes read or written. These events can be generated by by overriding the particular library function of interest (e.g. `malloc` for the number of bytes allocated) to count the measure of interest and to trigger sampling. Figure 2 presents a `malloc` replacement and section 2.5.2 illustrates how to override `malloc` without recompiling. It is also possible to combine the design presented here with a framework for *instrumentation sampling* to count metrics such as procedure calls [5]. In any event, we require the metric(s) under consideration be *monotonic* [15]–that is, the amount of metric consumed by the program increase monotonically over time.

2.2 Collecting Samples

A *call stack sample* is a list of instruction pointers representing the active procedures at an arbitrary point in the program. When a sample event occurs, the profiler computes a call stack sample by walking the stack from the event context to the root. In addition to the instruction pointers in the call stack sample, the profiler also records the stack pointer of each active procedure to properly distinguish recursive invocations. One can collapse recursive procedure activations [1] to limit the size the collected data, but in practice we have observed no problems with retaining the entire call stack.

Rather than storing each call stack sample, we represent the sampled data as a calling context tree (CCT) [1] in which the path from the root of the tree to a node corresponds to a distinct call stack sample seen during execution and a count at the deepest node keeps track of the number of times that path was seen. Because we store the instruction pointer for each unique call site rather than one identifier for each function, storing child nodes in the tree must be efficient. To strike a balance between lookup efficiency and space efficiency, we store the children of a node in a balanced tree.[4]

Abstractly, inserting a call stack sample into the CCT is done by traversing the tree downward from the root, creating new tree nodes and edges as necessary, and by incrementing the sample count at the tree node corresponding to the procedure activation on the top of the stack sample. An alternate approach to storing call stack samples was pursued by Hall [15], who stored each collected sample in its entirety in a simple list of samples. While this approach consumes more space than our CCT storage scheme, it preserves temporal information about when each sample was collected and may be preferable for some applications.

Doing a complete walk of the path for each sample is unnecessarily expensive, so we use memoization to limit the amount of memory touched at each insertion. We record the last call stack sample collected and pointers to the corresponding CCT nodes for each procedure activation in the sample. When a new sample is inserted, we find the last procedure activation where the new sample and previous sample agree. We then begin the insertion of the remaining portion of the new sample starting from the tree node associated with that procedure activation. By itself, this approach still requires a full traversal of the call stack.

2.3 Limiting Stack Traversal

To support high sampling frequencies it is necessary to make the the stack walking and data recording processes as efficient as possible. Performing a full traversal of the call stack at each sample event is expensive and unnecessary. For a given call stack sample, all procedure activations that existed when the previous sample was taken are already known, so there is no reason to unwind through these activations again. Some form of sentinel or high-water mark is needed to keep track of the unmodified prefix of the stack.

One method of implementing a sentinel the "sample bit" idea of Whaley [23]. This places a flag in each procedure activation indicating whether the procedure activation has been sampled in the past. New procedure activations are created with this bit clear. To collect a call stack sample the profiler only needs to walks procedure activations, marking as it goes, until a procedure activation with the sample bit set is found. At this point, the profiler can be certain that a common, already-sampled frame in 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.

One method for adding this flag is to place it in a low-order

[4]Profiling with an asynchronous sample source prohibits the use of `malloc`/`free`. Therefore, we prefer to use a data structure that does not require discarding old data as the amount of data grows.

bit of the return address. On systems with word aligned instruction addresses, the hardware often ignores these bits. If the hardware does not ignore the sample bit, the program under examination can be modified by inserting an instruction to do the masking before each procedure return. While this involves either modifying the compiler or the program binary (and all of its component libraries), this workaround retains all of the essential features of the sample bit approach.

Arnold and Sweeney [6] implemented a sentinel mechanism that works in environments where the hardware does not perform the necessary masking and modification of the program binary is undesirable. They replace the return address in each marked activation frame with the address of a hand crafted code fragment known as a *trampoline*. The overwritten return address is saved for later use. When a procedure activation that has been modified by call stack sampling tries to return, control is instead transferred to the trampoline. After performing its profiling functionality, the trampoline transfers control back to the application by looking up the most recently replaced then looks up the corresponding return address and jumps to that address. Using trampolines as sentinels incurs overhead when a sampled procedure activation returns, but does not require modifying the program binary.

The presence of executable code in the trampoline permits one further optimization to call stack sampling. Once the first previously-sampled frame is encountered the complete call stack is known, even though only a suffix has been walked for this sample. When using a sample bit, the previous prefix and the current suffix are concatenated and the resulting path is traversed in the recording structure. 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 of the shadow stack each time it is activated. Each node of the shadow stack contains a pointer to the corresponding node into the CCT. Thus, only that part of the CCT corresponding to the suffix needs to be traversed.

In contrast with Arnold and Sweeney, our implementation ensures that there is no more than one trampoline in any stack at any time. When the trampoline is invoked (i.e. when a sampled procedure returns), the trampoline installs itself above the procedure activation returned to. This enables the shadow stack to be maintained with a lower space cost, since only one return address need be stored at any time. When a sample event occurs, the trampoline is removed from its current position and is placed above the frame of the the interrupted procedure.[5] Figure 3 illustrates the movement of the trampoline in a profiled program.

2.3.1 Inserting the Trampoline

To insert the trampoline, the profiler must be able to determine the location of the return address at any arbitrary point in the program, including procedure prologues and epilogues. This information comes packed in small structures known as *procedure descriptors*. A procedure descriptor is a

[5]In practice, a sample event may occur while the trampoline is executing. Such samples are "unsafe", as unwinding the call stack is impossible at such an event. These samples, however, account for less than a tenth of a percent of the total number of samples taken in real applications.

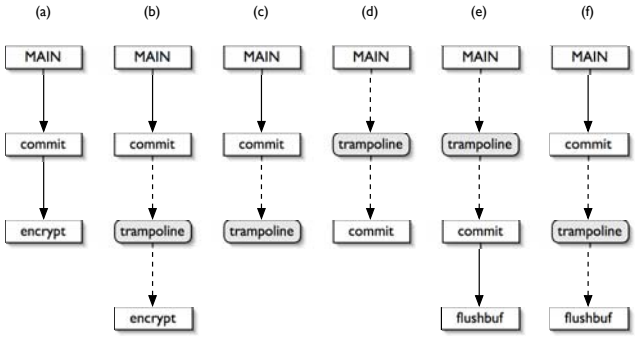

Figure 3: Operation of the profiler: (a) the program is running normally; (b) a sample event occurs and the trampoline is inserted; (c) `encrypt` returns to the trampoline; (d) the program resumes normal execution after the trampoline has moved itself up the call chain; (e) `flushbuf` is called and the trampoline is not moved; (f) a sample event occurs and the trampoline is moved to catch the return from `flushbuf`.

```
typedef struct _ra_loc {
  enum { REGISTER, ADDRESS } type;
  union {
    int reg;
    void **address;
  } location;
} ra_loc_t;

struct lox {
  ra_loc_t current, stored;
};
```

Figure 4: Definition of `struct lox`, which is filled in by a machine-dependent routine. The information is then used by the machine-independent portion of the profiler to perform the insertion of the trampoline.

data structure that describes key properties of a procedure, such as how much stack space it uses, whether it saves its return address in a register or on the stack, and so forth. While procedure descriptors are most often compiler-generated and used during exception handling, the profiler design presented in this paper is not dependent upon compiler-generated procedure descriptors. Procedure descriptors could be synthesized by analyzing the program binary (as well as any associated libraries) at load time or on the fly as the program runs.

Our profiler design hides the use of procedure descriptors behind a machine-independent interface. Figure 4 shows a C definition of a structure used to pass information about the location of the return address from the machine-dependent back end to the machine-independent front end. A `struct lox` must store two locations because a procedure may be interrupted in its prologue and may have not yet moved the return address to the location where it will reside during the body of the procedure.

Once this structure has been filled in, a simple case analysis is all that is necessary to insert the trampoline. There

is also an interface for removing the trampoline. Note that if the return address resides in a register, the context of the sample event will be modified. Modifying the context does not directly modify the register set of the machine; for asynchronous events, however, the modified context passed to the signal handler will be restored by the operating system. For synchronous events, however, the modified context will need to be restored explicitly via library routines.

2.4 Exposing Calling Patterns

Although the sampled events measure *where* costs were accumulated in the context tree, they do not tell the whole story. For example, a straightforward cost profile may indicate an application spent a significant fraction of its time in procedure f. However, the profile is unable to indicate whether each invocation of f is particularly expensive or whether f was simply called a large number of times. To understand the calling patterns, keep both node and edge counts in the CCT. Each node count corresponds to the number times the path from the root to the node was seen. Each time the trampoline is invoked, the edge count is incremented for the returning context. The effect is to accumulate counts on call chain edges that existed when a sample was taken, but that were removed before the next sample occurred.

The edge counts, which are maintained for every call path edge the profiler has seen, are equivalent to the "new call edges" counting strategy done by Whaley [23] and by Arnold and Sweeney [6]. The latter found this incrementing strategy unsuitable for their goal of approximating procedure call counts. Interpreting them for what they are–the number of times the edge was new when a sample was collected–provides an understanding of the amount of call stack activity between samples.

2.5 Messy Aspects of Real Programs

The abstract design presented above is sufficient if one is only interested in profiling simple examples in which there is a single stack in memory that is modified only by procedure calls and returns. Real applications, however, are more complex than this, using exception handling, dynamic loading, and multiple threads of control. Optimizing compilers can create more efficient calling sequences that break the assumptions of the design in the previous section. Several practical concerns must also be addressed before the profiler is suitable to use on real applications. In this section, we show how a portable design can deal with these complexities and concerns.

2.5.1 Dynamic Loading

In addition to permitting programs to link to shared libraries at compile time, modern operating systems also enable programs to load and unload shared libraries at runtime, a process known as *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 only collects program counters during stack unwind operations, there must be some provision made for mapping these program counters to the functions that contained them after the profile run has finished.

To provide for this mapping, the profiler identifies each sample taken with the shared libraries loaded at the time of the sample. 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 the samples for the new epoch and then samples are accumulated into this new tree.

While the loading of shared objects necessitates creating new epochs, new epochs can also be created for other reasons. For instance, a program that is aware of the profiler's existence could ask the profiler to collect new epochs at phase changes during execution: an epoch associated with initialization, an epoch associated with each distinct computation phase, and so forth. This mechanism enables the performance analyst to divide an application profile into distinct phases and also provides a method of temporally ordering the collected samples at a coarse level, delivering some of the benefits of storing complete samples, but without the space overhead.

2.5.2 Overriding Library Calls

Running the profiler in the same address space as the application allows the application to interfere with the profiler. For instance, the application might install its own handler for `SIGPROF`, which would replace the handler installed by the profiler and ruin the profiling run. To guard against such application actions, several functions in the standard C library are overridden with functions of the same name in the profiler's library. Since the profiler's library is linked after 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 malicious and should be allowed to complete–the application may wish to install a handler for `SIGSEGV`–and therefore the original functions must also be preserved. 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 performing the requested action. As an example, the profiler's version of `setitimer` checks to see whether the `ITIMER_PROF` timer is being set. If so, a warning message is printed, but no other action is taken; otherwise, the arguments are passed to the C library function via the appropriate captured function pointer. Callers of such overridden functions in the profiler always use the function pointers to avoid needless checking.

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 is 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.

2.5.3 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 prior to f in the current call chain, regardless of whether g was the immediate caller of f. Non-local exits occur primarily in programs that use exception handling; the standard C functions `setjmp` and `longjmp` are often used as a primitive form of exception handling in C.

If the trampoline is placed in the activation frame of a function that performs a non-local exit, then the trampoline will not be activated. Furthermore, the internal data structures of the profiler (e.g. the cached call stack) will not be updated and the profiler will be out of sync with the current execution of the program. Therefore, the profiler must intercept all non-local exit calls (see section 2.5.2) and modify its internal state to be consistent with the state to which the program is returning. The non-local exit can then be allowed to continue without knowing the profiler is active.

Functions that perform non-local exits are overridden to ensure that the necessary processing is done to maintain the internal state of the profiler. The obvious functions are `longjmp` and `siglongjmp`; for most environments, there is also a separate function called to transfer control to an exception that must be overridden as well. A single routine to handle the intricacies of non-local exits can be written and then called from the appropriate overridden functions. Each of these functions perform non-local exits by transferring control to a specified context. When a non-local exit occurs, the stack pointer, sp_c, is retrieved from the associated context and compared with the stack pointer at the top of the cached call stack, sp_r, to determine the action to take. In the following discussion, we assume that the program's stack grows downward.

If $sp_c <= sp_r$, then nothing needs to be done, as the context being restored either lies below the trampoline's context or is identical to the trampoline's context. Otherwise ($sp_c > sp_r$), the context being restored lies in a frame above the frame in which the trampoline is currently installed. In this case, nodes are popped from the cached call stack until a node is found that corresponds to the context being returned to. Such a node (frame) must exist due to restrictions on the manner in which `setjmp` and `longjmp` can be used.[6] The context being returned to 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 the functions being passed over had simply returned normally. After this handling, the non-local exit proceeds as normal with the (possibly altered) context.

Exceptions can be handled in much the same manner. Our profiler requires there exist two functions to serve as hooks into the exception handling process[7]; these functions will be overridden by the profiler. The first function is called to initiate a search for an appropriate exception handler. In the profiler's version, a flag is set signifying that an exception is being processed and no stack samples should be taken, nor should the trampoline be moved. After this, the original routine is called. The second function is called after a handler is found to transfer control to the handler; this transfer of control is a non-local exit similar to `longjmp`. When the profiler overrides this function, the trampoline is (potentially) moved according to the above process, the flag is reset, and control then resumes in the handler.

[6] `setjmp` and `longjmp` can be used to construct a co-routine based user-level threading system. We do not handle this fully general case of `setjmp` and `longjmp`; any use of threading will be handled by a multithreaded version of the profiler (see section 2.5.5).

[7] We assume the widely-used "two-phase" exception handling model [10].

2.5.4 Optimized Calling Sequences

For certain function calls, the compiler may generate more efficient code than for the general case. If a function call to g is the last thing function f does before returning, then g is said to be in *tail position* and the call to g is a *tail call*. If the compiler recognizes this special case, *tail call optimization* can be performed: f will branch directly to g and g will reuse f's stack frame. Our profiler and its associated trampoline, however, relies on functions returning "normally"–through a "jump to register" instruction rather than direct branches. If f was to return through the trampoline but instead tail calls g, then g will be returning through the trampoline, since g merely reuses f's return address. Furthermore, the profiler does not know where the trampoline address is located (the location where f saved the return address is not identical with g's location for the same) and is unable to remove the trampoline at the next sample event unless special care is taken for this case.

Consider a function f in which the trampoline is installed; f then tail calls g. There are three possibilities as to what will happen from the perspective of the profiler: g will return before the next sample event, a sample event will occur in g, or a sample event will occur in a procedure directly or indirectly invoked by g. In the first case, the trampoline will record a return from f and install itself in f's caller. This is exactly the same set of actions that would have occurred if f had returned normally. No problems arise in this scenario.

In the latter two cases, the profiler will discover the tail call by noticing that the location that was supposed to contain the address of the trampoline does not. The profiler must then reconcile its cached call stack, which is no longer a prefix of the actual call stack, with the actual call stack of the program. These cases are handled during the collection of the partial stack sample for the event. As the stack is being walked, the profiler checks if the instruction pointer of the current context is equal to the address of the trampoline. If so, then a tail call occurred.

Three steps are then necessary. First, the instruction pointer of the current context is replaced with the saved return address from the last insertion of the trampoline. Second, the cached call stack is popped to reflect that f no longer exists in the call stack and the current pointer into the collected CCT is updated appropriately. Finally, by consulting the procedure descriptor for the previous context's instruction pointer, the rogue trampoline can be removed from its location, whether on the stack or in the context. These steps preserve the invariant that each thread of control has at most one trampoline active at any time. After these steps, the tail call has been handled and the partial stack sample for the sample event has been collected.

2.5.5 Threaded programs

When multiple threads are involved in a program, each thread maintains its own trampoline, cached call stack, and CCT. We assume the POSIX thread model in the discussion below, but the concepts should be easily adaptable

to other threading systems. To create the private profiling state for each thread, we override `pthread_create` to discover thread creation. Our profiler supports monitoring any thread created through `pthread_create`, whether it is a user-level thread or kernel-level thread. Our version of `pthread_create` creates a small structure containing the user-passed parameters for the thread, passing the real `pthread_create` function this structure and an alternative initialization function. This alternate function initializes the profiling state in the context of the created thread rather than the parent thread.

3. TRU64/ALPHA IMPLEMENTATION

We have implemented the above design for the Tru64/Alpha platform. In this section, we give a brief overview of how the key parts of our profiler, `csprof`, were implemented. However, there were some parts of our design that could not be adequately expressed on our target platform. Below we indicate several inconsistencies and how we chose to work around those inconsistencies.

3.1 Basic Profiler Infrastructure

Tru64 provides a well-specific interface for accessing procedure descriptors and unwinding the stack in `libexc`. Our idealized procedure descriptors described in the earlier sections appear as two structures on Tru64: *code range descriptors*, which attribute specific properties to regions of code, such as whether a region is used for exception handling and where a region's entry point is; and *runtime procedure descriptors*, which encompass the remainder of the functionality. Walking the stack is performed with `exc_virtual_unwind`, which modifies a provided context to represent the previous procedure invocation. `libexc` also provides the necessary "hooks" into the exception handling subsystem: `exc_dispatch_exception` is overridden to notify the profiler of when an exception is being processed, and `exc_continue` is called to perform the non-local exit when an appropriate handler has been located. Our implementation is driven by `SIGPROF` signals generated by a system interval timer.[8]

3.2 Handling Procedure Epilogues

Procedure descriptors on the Alpha do not provide enough information to indicate whether a particular instruction pointer falls in the epilogue region of a function. This information is implicitly provided by the ABI conventions of the platform [8]. Unfortunately, the compilers we used for our experiments do not always generate ABI-conformant code. To insert the trampoline, the profiler must be able to determine the difference between body code and epilogue code. Therefore, a heuristic must be used to determine when the procedure is executing its epilogue.

To distinguish between body and epilogue code, we look backwards several instructions from the current instruction pointer, searching for an instruction that deallocates the function's stack frame. Searching for an instruction that loads the return address from the stack frame does not work in general because some paths through the function may not load the return address from the stack frame. If this frame-deallocating instruction is found, then the function is in its epilogue and the trampoline must be inserted directly into the register set. Otherwise, the function is executing its body code and the return address is found on the stack. However, an extra step is needed to handle those paths where the return address is not loaded from the stack (and also those cases where we have loaded the return address, but not yet deallocated the stack frame). We record the return address as being on the stack in a `struct lox` as indicated in section 2.3.1, but we also check the contents of the return address register. If the address there and the address found on the stack match, then we also insert the trampoline into the register set directly. This extra step is done in the machine-dependent Alpha backend. Doing this ensures the trampoline is correctly inserted to catch the return from the function in which it is placed.

[8]Hardware performance counter overflow is not exposed to userspace on Tru64.

4. EXPERIMENTAL RESULTS

This section presents experimental results for the SPEC CPU2000 benchmark suite on a Tru64/Alpha system. We compiled the benchmarks with optimization using HP's system compiler: C benchmarks were compiled with `cc`, version 6.4-009, using the `-fast` flag; C++ benchmarks were compiled with `cxx`, version 6.5-014, using the options `-tune host -O2`; and Fortran benchmarks with `f90`, version 5.5-1877, using the flags `-tune host -O5`. For our experimental platform, we used a lightly-loaded quad processor ES45 with 667MHz Alpha EV67 processors and 2GB of main memory running Tru64 5.1A. To produce the numbers in our results, we ran each program five times and report the mean of the five runs. For `gprof` runs we added the `-pg` flag to the compilation command line for each application; instrumentation was not added to libraries for these runs. For consistency with `gprof`, all `csprof` experiments sampled the program counter every millisecond.[9] The C compiler that we used did not generate correct procedure descriptors for `176.gcc` and so we were unable to use `csprof` to profile `176.gcc`. Table 2 summarizes our findings.

4.1 Overhead

As expected, `gprof`'s overhead, shown in column three, is generally high and is proportional to the number of calls executed by the program, shown in column four.[10] In contrast, `csprof`'s overhead, found in column five, remains relatively constant regardless of the call volume. Even for `lucas` and `swim`, which make very few calls, `csprof`'s overhead is comparable to `gprof`'s. `csprof` performs comparably to `gprof` in the average case while performing significantly better in the worst case. The size of `csprof`'s data files ranged between 30KB and 12.5MB, with a median of 232KB. Since the data file is a serialized dump of the in-memory structure to disk, this data also serves as an estimate for the memory required by `csprof`. Memory usage is moderate for most programs, especially considering the high sample rate.

`equake` represents a curious case: it makes over a billion calls, comparable to other benchmarks such as `mesa` and `wupwise`, yet has a miniscule overhead when profiled with `gprof`. Examining the `gprof` output indicates that three

[9]Using a lower sampling for `csprof` would enable us to reduce the call path collection overhead of `csprof` by an arbitrary amount.

[10]For benchmarks that use multiple invocations of the benchmark program for each run (`gzip`, `vpr`, `gcc`, `eon`, `perlbmk`, `vortex`, and `bzip2`), we have summed the call counts over all invocations of the benchmark.

Benchmark	original time (seconds)	gprof overhead (percent)	gprof calls	csprof overhead (percent)
Integer programs				
164.gzip	479	53	1.960×10^9	4.2
175.vpr	399	53	1.558×10^9	2.0
176.gcc	250	78	9.751×10^8	N/A
181.mcf	475	19	8.455×10^8	8.0
186.crafty	196	140	1.908×10^9	5.1
197.parser	700	167	7.009×10^9	4.6
252.eon	245	263	1.927×10^9	3.4
253.perlbmk	470	165	2.546×10^9	2.5
254.gap	369	39	9.980×10^8	4.1
255.vortex	423	230	6.707×10^9	5.4
256.bzip2	373	113	3.205×10^9	1.1
300.twolf	568	59	2.098×10^9	3.0
Floating-point programs				
168.wupwise	353	85	2.233×10^9	2.5
171.swim	563	0.17	2,401	2.0
172.mgrid	502	0.12	59,177	2.0
173.applu	331	0.21	219,172	1.9
177.mesa	264	67	1.658×10^9	3.0
178.galgel	249	5.5	1.490×10^7	3.2
179.art	196	2.1	1.11×10^7	1.5
183.equake	549	0.75	1.047×10^9	7.0
187.facerec	267	9.4	2.555×10^8	1.5
188.ammp	547	2.8	1.006×10^8	2.7
189.lucas	304	0.30	195	1.9
191.fma3d	428	18	5.280×10^8	2.3
200.sixtrack	436	0.99	1.03×10^7	1.7
301.apsi	550	12	2.375×10^8	1.6

Table 2: Results from our experiments. The columns "gprof overhead" and "csprof overhead" show a percentage overhead relative to the original execution time column. (An overhead of 100% indicates that the monitored execution took twice as long.) We were unable to obtain results for 176.gcc and csprof due to a bug in the version of the C compiler we used; this bug generated incorrect procedure descriptors and therefore caused csprof to crash.

functions account for over 99.9% of the calls and that these functions are short–exactly the kind of functions on which we would expect gprof to incur high overhead. However, since the compiler had access to both the application code and the inserted profile code, we suspected that the functions were inlined and the compiler recognized the profiling code to be loop invariant. The compiler then moved the profiling code out of the inner loop–maintaining the call counting provided by the code, but drastically reducing its cost. To test this theory, we used Tru64's hiprof tool, which adds gprof-style instrumentation to application binaries, to instrument the unprofiled, optimized binary and used the instrumented binary for a test run. Profiling in this manner increased the runtime to 956 seconds, an overhead of 74%, which is much closer to our expectations. This result suggests that our comparison can be considered to be against the "best case" of gprof–that is, the case where the compiler has the opportunity to optimize the added instrumentation.

	csprof	gprof
Integer programs		
Minimum	0.74	7.6
Median	2.9	15
Mean	8.0	23
Maximum	51	120
Floating point programs		
Minimum	0.37	0.33
Median	3.6	2.4
Mean	5.0	4.1
Maximum	18	15

Table 3: Distortion caused by csprof and gprof relative to "base" timing measurements given by DCPI.

4.2 Accuracy

Simply comparing overheads, however, does not indicate whether one profiler is to be preferred over another; a comparison of the quality of the information provided by each profiler should also be done. We consider here the distortion of cost to individual functions in a profiled program p, meausured by the following formula:

$$\sum_{f \in functions(p)} |P_x(f) - P_{dcpi}(f)|$$

We measured flat profiles of each benchmark program using DCPI [3], a low-overhead sampling profiler driven by hardware performance counters. This information is used in the above equation as a close match to the actual behavior of the program. $P_x(f)$ returns the percentage of time consumed by a function 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 DCPI.

A summary of our accuracy comparison is given in Table 3. On the integer benchmarks, csprof is markedly better than gprof, delivering lower distortion on 10 out of the 11 benchmarks. On the floating point benchmarks, many of which execute relatively few procedure calls, the situation is somewhat more mixed; csprof delivers lower distortion than gprof on four out of the 14 benchmarks. However, on those benchmarks where gprof delivers lower distortion, the average difference between csprof's distortion and gprof's distortion is only 0.47.

5. CURRENT DIRECTIONS

One problem for which no adequate solution currently exists is the effective presentation of data gathered by context-based profilers for large, complex programs. An obvious idea used in many tools is to present a tree or directed graph where nodes represent functions, or function invocations, and edges represent calls. This approach has been used in several tools and it is effective for small, simple programs, but it does not scale well. There are several packages available that, when supplied with specifications for the nodes and the edges, will automatically draw a "pretty" graph for a human viewer [11, 19]. Combined with controls to limit the drawn graph to "hot" paths, this method can be quite useful. While feasible for interactive use as well, this method is probably not the best choice from a user interface perspective. If the user wishes to focus on a subset of the nodes in

the graph, the underlying graph layout package will probably have to recompute the layout of the new graph. Doing this has a high probability of rearranging nodes and edges, confusing the user in the process.

The most compelling idea for limiting the size and complexity of displayed graphs comes from Hall's work on call path refinement profiles [14]. In Hall's scheme, focusing on particular paths in the graph is accomplished primarily by *call path refinements*, which are filters that specify the paths for which performance data should be computed and displayed. For example, when presented with the call tree of a large scientific code, the user might display only the paths that contain calls to synchronization routines. Next, the user might further refine the display to only those paths containing calls to the differential equation solver. This new filter is only applied to those paths containing calls to synchronization routines, enabling the user to effectively zoom in on a very specific area of the program. BOTTLENECKS by Ammons *et al.* [2] is a tool that applies Hall's ideas to any profile that associates metrics with executable call paths. The tool suggests paths that may yield performance improvements, keeps track of the necessary bookkeeping during analysis, and supports comparing profiles to pinpoint performance problems. Interfacing profiles collected by `csprof` to BOTTLENECKS would be trivial.

HPCToolkit (formerly called HPCView [17]) is a collection of tools for correlating multiple flat performance data profiles with source code. Although the input data consists of flat profiles, for analysis and presentation HPCToolkit aggregates the data hierarchically using the static structure of the program (load modules, files, procedures, loop nests, and statements). `hpcviewer` is the performance browser component of HPCToolkit. It encourages a top-down approach to browsing the data by providing mechanisms for manipulating the programmer/analyst's view of the hierarchy as a tree while displaying a performance data table for those elements of the tree that are currently visible. This combination has proven to be a very productive way of presenting the data because it enables the user to start with the whole program and to quickly explore the significant parts of the program by adjusting the refinement of the static tree.

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

While the details of unwinding and determining where to place the trampoline is architecture-specific, most of `csprof` is platform-independent and we are exploring ports to other architectures. A prototype x86-64 port has been completed and we are currently making it more robust.

6. CONCLUSIONS

Modern modular programs rely heavily on libraries for much of their functionality. Knowledge of the calling context of performance bottlenecks is a crucial piece of information in understanding how to address those bottlenecks. Instrumenting all the functions of an application is one way to provide extensive context information, but the overhead of executing the added instrumentation on every call is burdensome.

In this paper, we describe an efficient and accurate method to collect context information using stack sampling. Marking a stack frame with a sentinel (i.e. a trampoline address) enables us to reduce the run-time overhead of stack sampling. In addition, unlike previous call path profilers, we are able to associate edge counts with call path edges in the sampled call graph to provide the analyst with a measure of call stack activity between events. All of this is done efficiently without requiring special compiler support or modification of the program's code. Our experiments have shown that the call stack sampling technique described in this paper provides more accurate profiles and incurs lower overhead on call-intensive codes than a conventional call path profiler that adds instrumentation at every call.

Acknowledgments

Nathan Tallent wrote most of the original `csprof` code for the Intel Itanium architecture. This material is based on work supported by the National Science Foundation under subaward No. 768ET11034A from the University of Illinois. and by the the Los Alamos Computer Science Institute under Department of Energy Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory.

7. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1997.

[2] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy. Finding and removing performance bottlenecks in large systems. In *Proceedings of the 2004 European Conference on Object-Oriented Programming*, pages 172–196, 2004.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 1–14. ACM Press, 1997.

[4] Apple Computer. Shark. `http://developer.apple.com/performance/`.

[5] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[6] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report 21789, IBM, 1999.

[7] A. R. Bernat and B. P. Miller. Incremental call-path profiling. Technical report, University of Wisconsin, 2004.

[8] H.-P. Company. Calling standard for Alpha systems. `http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V51B_HTML/ARH9MCT%E/TITLETXT.HTM`. 29 April 2005.

[9] T. C. Conway and Z. Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical Report 24, University of Melbourne, Australia, 2001.

[10] S. J. Drew, K. J. Gough, and J. Ledermann. Implementing zero overhead exception handling. Technical Report 95-12, Queensland University of Technology, 1995.

[11] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 29(5), 1999.

[12] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[14] R. J. Hall. Call path refinement profiles. In *IEEE Transactions on Software Engineering*, volume no. 6, 1995.

[15] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the USENIX Summer Technical Conference*, 1993.

[16] Intel Corporation. Intel vtune performance analyzers. http://www.intel.com/software/products/vtune/.

[17] J. Mellor-Crummey, R. Fowler, G. Marin, and N. Tallent. HPCView: A tool for top-down analysis of node performance. *The Journal of Supercomputing*, 23:81–101, 2002. *Special Issue with selected papers from the Los Alamos Computer Science Institute Symposium.*

[18] C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Software: Practice and Experience*, 18(5), 1988.

[19] G. Sander. Graph layout through the VCG tool. In R. Tamassia and I. G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 194–205, Berlin, Germany, 10–12 1994. Springer-Verlag.

[20] M. Spivey. Fast, accurate call graph profiling. *Software: Practice and Experience*, 34(3):249–264, 2004.

[21] D. A. Varley. Practical experience of the limitations of gprof. *Software: Practice and Experience*, 23(4):461–463, 1993.

[22] O. Waddell and J. M. Ashley. Visualizing the performance of higher-order programs. In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 75–82. ACM Press, 1998.

[23] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Java Grande*, pages 78–87, 2000.