Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Call path profiling for unmodified, optimized binaries

Nathan Froyd, Nathan Tallent, John Mellor-Crummey, and Rob Fowler
Department of Computer Science
Rice University
{froydnj,tallent,johnmc,rjf}@cs.rice.edu

Abstract

Understanding the performance of today's large and complex programs requires a new generation of profiling tools that attribute costs to the full calling contexts in which they are incurred. We describe a new call path profiler for optimized code and the changes needed in GCC and other parts of the GNU tool chain to support such tools.

Although `gprof` has long been the standard for call graph profiling in the GNU toolchain, it suffers from several shortcomings. First, in modern object-oriented programs, costs must be attributed to full calling contexts because the cost of each call can be context dependent; `gprof` ignores this issue. Second, `gprof` relies upon instrumentation in procedure prologues to collect performance data. This imposes four costs: (1) recompilation is required; (2) nested instrumentation distorts the measurements; (3) instrumentation overhead can be significant; and (4) the instrumentation interferes with aggressive compiler optimization. We have developed a call-path profiler that avoids all of these shortcomings. To measure the performance of unmodified, fully-optimized binaries, we use stack unwinding and sampling to attribute costs to calling contexts and to collect frequency counts for call graph edges. Unlike `gprof`, `csprof` accurately attributes context-dependent costs. Furthermore, experiments with the SPEC CPU2000 benchmarks show that our new profiling strategy is significantly more efficient than `gprof` as well.

Profilers and other tools that rely on stack unwinding must correctly handle events that occur at any point in the execution. This requires more comprehensive unwinding information than what compilers already record to support C++ exceptions. In this paper, we describe changes made to GCC to record the additional information necessary. We present experimental results on the x86-64 platform that show our profiler has low overhead and that the additional unwinding information it requires is of modest size. To cope with compilers that don't record comprehensive unwind information, it is possible to recover this information using binary analysis of executables. We discuss modifications to `binutils` needed to support this and related analyses.

1 Introduction

The gap between peak and typical performance on modern microprocessor architectures has been growing with each new generation of processors. Today, only carefully tuned applications achieve a substantial fraction of peak performance. While optimizing compilers improve application performance, compiler optimization often fails to deliver much of the improvement possible. As a result, much of the burden of performance tuning falls to application developers. Good performance tools are essential to help developers determine where they should invest their tuning effort.

Understanding where an application spends its time is only the first step toward tuning. The next steps are determining whether this is a symptom of inefficiency and understanding how inefficiency arises. Sometimes the answers are local, e.g., a loop iterating over a large amount of data doesn't utilize the memory hierarchy effectively. Often, answers are elusive. Is the time spent in a procedure the result of inefficiency in the procedure, or the result of the procedure being invoked frequently? An understanding of the contexts in which costs are incurred is vital for analyzing object-oriented abstractions, component-based software, and instantiations of templates for data and computation. Can the costs associated with using a particular abstraction, e.g., a set, be reduced by picking a different implementation? Questions such as this may have multiple answers: different choices may be appropriate for different instances of an abstraction within a program. For instance, a bit vector can be a good implementation where a dense set is needed, but other representations are preferable for sparse sets. To tune a program effectively, developers must know the contexts in which costs are incurred to choose among the myriad possibilities for tuning.

For over two decades, the approach pioneered in `gprof` [7] has been the standard for acquiring contextual information (in the form of call graphs) to interpret costs incurred. `gprof` inserts instrumentation in procedure prologues to log procedure entry and increment a count associated with (callsite, callee) pairs. A critical shortcoming of `gprof` is that it assumes that the cost of a function call is independent of its calling context. The cost of a function or method call can vary widely between object, component, or template instances and `gprof` lacks the ability to help pinpoint such differences to guide application tuning.

Using `gprof` has four additional shortcomings. *First,* `gprof` relies upon information collected by instrumentation in procedure prologues. With the GNU toolchain, adding this instrumentation requires recompilation of the program and all the non-standard libraries that it uses.

Second, executing instrumentation code in each procedure call dilates execution time. This can preclude the use of `gprof` on large production runs. We have observed a 3-14x slowdown on different systems of an execution of a synthetic call-intensive "torture test" written to showcase this problem [6]. Execution time dilation is not just a theoretical concern. Section 4 describes experiments with the SPEC CPU2000 integer benchmarks [12], which show that `gprof` instrumentation increases execution time by 82% on average. Object-oriented code with small methods is especially sensitive to dilation.

Third, the instrumentation in nested procedure calls dilates the measured cost of each procedure, thus introducing a systematic measurement error. This dilation disproportionately inflates costs attributed to small procedures. An analysis of how the fraction of time attributed to each function by `gprof` differs from that attributed by DCPI [2]—a well known, flat profiler with very low overhead—showed that for the SPEC CPU2000 integer benchmarks `gprof`'s measurements were distorted by 23% on average [6].

Fourth, `gprof`'s instrumentation-based approach precludes some compiler optimization.

For instance, in the GNU toolchain, gprof instrumentation requires frame pointers and thus, code measured with gprof cannot be fully optimized; this contributes to measurement distortion. Instrumentation-based approaches also inhibit inlining and/or post-inlining optimizations.

To avoid the shortcomings of gprof, we built a new profiler csprof [6]. Rather than relying on instrumentation in procedure prologues, csprof uses call stack unwinding to attribute samples to calling contexts and associate frequency counts with call graph edges. This approach has several benefits. First, csprof can be used to profile unmodified, fully-optimized programs without recompilation. Second, csprof records information about the full calling context rather than just call graph edges. Third, csprof does not incur overhead of instrumentation on every function call and thus neither incurs high overheads nor does it systematically distort measurements.

For csprof (or any other call-path profiler based on stack unwinding) to work properly on fully-optimized code, it must be able to unwind the call stack at *any* point in a program's execution, even when no frame pointer is used. Successfully unwinding from an event that occurred during a procedure epilogue requires precise information about the machine state at each point in the epilogue. This requires more information than is needed, *e.g.,* to unwind the stack in for C++ exception handling. In this paper, we describe the changes we made to GCC to emit this extra information, the size of this additional information, and we present results of experiments with csprof using this information on the x86-64 platform. In addition to changes to GCC, we believe that modernization of binutils to better support performance tools is in order.

The rest of this paper is structured as follows. Section 2 presents the high-level design of our call-path profiler csprof. Section 3 describes modifications we made to GCC to enable call stack unwinding of unmodified, fully-optimized code at any point in a program's execution. Section 4 compares the overhead and accuracy of csprof with that of gprof and reports how our changes to GCC affect the size of the unwind information it records. Section 5 briefly describes related work on open-source tools for call stack profiling. Section 6 describes shortcomings of the binutils library for performance tools and offers some suggestions for improving it. Section 7 presents our conclusions, ongoing work, and plans for the future.

2 Profiler Design

Our requirement for csprof was that it be easy to use with large, modern applications. Hence, it works on dynamically linked, optimized, unmodified binaries.[1] To initiate profiling, csprof instructs the dynamic loader (via the LD_PRELOAD environment variable or equivalent) to pre-load csprof's profiling library. The library's initialization routine allocates and initializes profiler state and then initiates profiling. The library's finalization routine halts profiling and writes the profiler state to disk for later analysis.

csprof works with both asynchronous and synchronous events. Asynchronous events are not initiated by direct program action. They arise from interrupts triggered by the UN*X interval timer and/or hardware performance counter traps. Asynchronous events are monitored by setting up a signal handler to log each event and associate it with its call-path context.

[1]Profiling statically-linked applications is possible, but outside the scope of this paper.

Synchronous events are generated via direct program action. Examples of interesting events for synchronous profiling are memory allocation, I/O, and interprocessor communication. For such events, one might record bytes allocated, written, or communicated, respectively. Monitoring synchronous events typically involves having `csprof` use dynamic loading to override the relevant library routines and then to log information as appropriate when a monitored routine is called.

When synchronous or asynchronous events occur, `csprof` records the *full calling context* for each event. A calling context collected by `csprof` is a list of instruction pointers, one for each procedure frame active at the time the event occurred. The first instruction pointer in the list is the program counter location at which the event occurred. The rest of the list contains return addresses for each of the active procedure frames. We retain stack pointers as well to distinguish between recursive invocations. We have not observed excessive space requirements when retaining entire call stacks; if the storage of samples were to become a concern, we could collapse calling contexts for recursive procedure invocations [1] or record only a suffix of full contexts.

We store samples and their calling contexts in a *calling context tree* (CCT) [1]. In a CCT, the path from each node to the root of the tree represents a distinct calling context. Counts associated with events (e.g. cache misses, microseconds, bytes allocated) are attached to each node in the tree to associate the metrics with the calling context in which they were recorded.

Using a sentinel to limit unwinding. To ensure good statistical coverage of profiled code, one must collect a large number of samples, either by measuring over a long interval, or by using a high sampling rate. In either case, it is desirable for the unwinding and sample recording process be as efficient as possible. Performing a full unwind of the call stack at each sample event has the potential to be costly. To avoid this, we use a sentinel to mark the stack frames in existence when a sample event occurs. When processing the next sample event, we don't need to unwind frames below the sentinel since they have already been recorded.

We use dynamic execution state modification ("stack surgery") to implement stack sentinels in a general way for systems on which we control neither the compilers nor the calling conventions and stack frame layout. We mark a procedure frame with a sentinel by replacing its return address with the address of a *trampoline* function. `csprof` stops unwinding when it finds the trampoline as the return address of a stack frame. After the context of each sample event is recorded, if the frame currently marked by the sentinel is no longer the top stack frame, `csprof` unmarks that frame and marks the top stack frame instead. When a marked procedure returns through the trampoline, the trampoline moves the sentinel to mark the caller's frame before transferring control back into the caller.

In addition to using the sentinel to limit the depth of stack unwinding, `csprof` also memoizes previously inspected calling context by keeping a stack of pointers to the corresponding nodes in the CCT. Inserting a new sample in the CCT thus begins at the node corresponding to the sentinel frame rather than the root of the CCT. This reduces the number of memory references needed to record each sample.

Exposing calling patterns. Besides knowing the full calling context for each sample event, it is useful to know how many unique calls are represented by the samples recorded in a calling context tree. This information enables a developer interpreting a profile to determine whether a procedure in which many samples were taken was doing a lot of work in a few calls or a little work in each of many calls. This knowledge in turn determines where optimizations should be sought: in a function itself or its call chain. To collect edge frequency counts, we increment an edge traversal count as the program returns from each stack frame active when a sample event occurred. We do this by having the trampoline increment a "return count" for the procedure frame marked by the sentinel as it returns. A more detailed description of this strategy can be found elsewhere [6].

Handling the complexity of real programs. The high-level design for `csprof` that we have described thus far suffices only for profiling simple programs with a single stack in memory that is modified only by procedure calls and returns. Real programs are often more complicated, featuring dynamic loading and unloading of code, register frame procedures, exception handling (including `longjmp`), and multiple threads of control. Many compilers, including GCC, generate tail calls for certain classes of function calls; these need to be handled specially by `csprof`. In addition, it is possible to receive events during execution of the trampoline or the sampler. The details of dealing with these complexities are outside the scope of this paper and are described elsewhere [6].

3 Adding Support to GCC

Our profiler design requires that the stack can be unwound from arbitrary PC locations during a program execution. This is a stricter requirement than the previous client for stack unwinding, namely, C++ exception handling. Exception handling requires only that the stack can be unwound from within a procedure body and never requires unwinds from within a procedure's prologue or epilogue(s). Most modern implementations of C++ use table-driven exception handling [5], where a small table describes the effects of instructions within a range of PCs. The effects recorded are only those that are necessary to unwind the stack properly, such as adjustments to the stack pointer, movement of the return address, in addition to register saves and restores. For example, GCC uses the DWARF2 format [13] for unwind tables. In practice, providing support for unwinding from within prologues is no more difficult than providing support for unwinding from within the body of a procedure.

Supporting unwinding from within epilogues is more complicated due to the way DWARF2 unwind information is interpreted. Conceptually, DWARF2 unwind information forms a table. For every instruction in a procedure, this table encodes rules for obtaining 1) the VMA of the "canonical frame address" (the stack pointer upon entrance to the procedure) and 2) the value on procedure entry of (callee-saved) registers. The rules are encoded as a byte-coded instruction stream. If rules for every instruction were included, the space consumed by the unwind information would be considerable. However, the matrix is generally sparse: the rules for many of the table rows are identical to those for the previous row. Therefore, only a limited number of instructions must have their effects encoded and `advance_loc` byte codes can be used to skip redundant rows. To compute the unwind information for any given instruction, the cumulative effects of all previous instructions in the enclosing procedure must be considered. Thus, to unwind the activation record from a given instruction, one iterates through the sequence of byte codes, applying effect rules until an address greater than the current instruction's VMA is found.

Because of branch and jump instructions, there

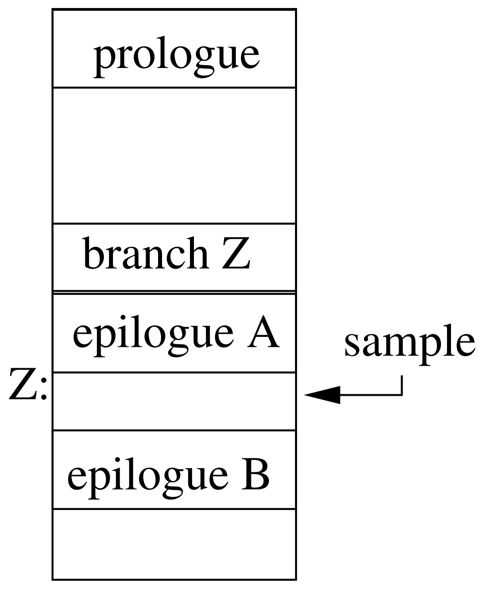

Figure 1: A routine with two epilogues, one of which is interior. Before the interior epilogue (A) there is a conditional branch instruction. We wish to begin unwinding from the sample point.

may be an inconsistency between this linear accumulation of instruction effects and the instructions that are actually executed. Consider an interior epilogue as depicted in Figure 1.[2] A linear scan of instruction effects prior to the sample point "ignores" the conditional branch before epilogue A and leads to the erroneous conclusion that all callee-saved registers have been restored and the current stack frame de-allocated as the scan progresses through the effects of instructions in epilogue A! DWARF2 provides two byte codes — `remember_state` and `restore_state` — that should "parenthesize" interior epilogues. When a DWARF2 unwinder sees a `remember_state` byte code, it conceptually pushes a copy of the current table onto a stack; when it sees a `restore_state`, it restores the table's state with the top of the stack and issues a pop.

[2]A compiler may generate an interior epilogue to improve instruction cache locality.

Our modifications to properly handle unwinding from within epilogues fall into three categories:

- Tagging frame-related epilogue instructions as such in the x86-64 back end;
- Support in the DWARF2 emitter for epilogues;
- Handling multiple epilogues with care.

3.1 Tagging instructions

The DWARF2 unwinding process is based on knowing the effects of "frame-related" instructions, namely, instructions that modify the stack pointer and/or save/restore registers. GCC's back end tags individual RTL pieces with the `FRAME_RELATED_P` tag to indicate their frame-relatedness. Adding the necessary `FRAME_RELATED_P` bits to the x86-64 back end was straightforward. As we worked on the DWARF2 emitter, however, we discovered two additional things that needed to be done.

When the DWARF2 emitter finds a `PARALLEL` RTX that is marked as `FRAME_RELATED_P`, it always processes the first of the child RTXs as if it is a `SET` and then processes subsequent RTXs if they are `SET`s and have the `FRAME_RELATED_P` flag set as well. In our initial attempt tagging, we only set `FRAME_RELATED_P` on the outer `PARALLEL` RTX, but not on any of the children of the node. This caused problems with instructions such as `pop`, which are naturally represented as `PARALLEL` RTXs. When correcting this, we also found that we had to modify the RTL definition of the `leave` instruction to be amenable to the DWARF2 emitter.

3.2 DWARF2 support

We implemented DWARF2 support in two steps. The first step was to fix GCC's DWARF2 emitter to handle instruction patterns commonly found in epilogues. The DWARF2 emitter knew how to handle register-to-memory moves, for example, but was ignorant about how to process memory-to-register moves. When processing the latter, we not only recorded effects on the stack pointer, but also encoded the DWARF2 byte codes indicating which registers had been restored. Extensions to the emitter to handle negative adjustments to the stack pointer and restoring the stack pointer from the frame pointer were also necessary.

The second step was to provide support for emitting the `remember_state` and `restore_state` byte codes. At first blush, this seemed simple enough: when processing a block, if an `EPILOGUE_BEG` note is encountered, emit a `remember_state` byte code. At the end of the same block, emit a `restore_state` byte code. However, we optimized this process so that when an epilogue block is the last block in the function, no `restore_state` byte code is necessary. This approach is similar to the strategy used by the IA-64 back end.

3.3 Multiple epilogues

In GCC's internal representation, epilogues are implicit, beginning with an `EPILOGUE_BEG` note and lasting until the end of the basic block. However, correctly handling epilogues requires that the relevant notes are actually inserted, which was not happening in two cases. The first was when a tail call, a.k.a. "sibcall," epilogue was generated, which was simple to fix. The second was when an epilogue block was duplicated; the `EPILOGUE_BEG` note was not duplicated as one might expect. Furthermore, when the initial non-sibcall epilogue is generated for a function, its instructions are inserted into a special `epilogue` array for use by the instruction scheduler. We found it necessary to insert the instructions of duplicated epilogues into this array to correctly handle repositioning of `EPILOGUE_BEG` notes. Additionally, when a duplicated epilogue was removed (e.g. by cross-jumping), `can_delete_insn_p` failed to indicate that the `EPILOGUE_BEG` note should be removed as well, presumably because multiple `EPILOGUE_BEG` notes were not a concern prior to our modifications.

Even when we ensured that `EPILOGUE_BEG` notes were consistently generated and duplicated as necessary, we found that certain passes of the compiler, particularly the instruction scheduler, assumed that only a single `EPILOGUE_BEG` note existed. To avoid littering the instruction scheduler with special cases for notes, any notes other than basic block boundary markers are removed prior to instruction scheduling and replaced afterwards. Since epilogue instructions may move across basic block boundaries, the instruction scheduler replaces `EPILOGUE_BEG` notes (`reposition_prologue_and_epilogue_notes`) by looking for the first instruction that is located in the `epilogue` array and positioning the note prior to that instruction. This is a flawed strategy when multiple epilogues are present.

We re-wrote the repositioning of the epilogue notes to correctly handle multiple epilogues. Our revised algorithm works as follows: We scan forward through all instructions,[3] looking for either an `EPILOGUE_BEG` note or an instruction that is contained in the "epi-

[3]Comments in `reposition_prologue_and_epilogue_notes` indicated that we cannot depend on the basic block structures maintained by GCC at this point.

logue" array mentioned earlier. When we find an `EPILOGUE_BEG` note, we move it to immediately before the next epilogue instruction that we find and skip to the end of that basic block. If instead we find an instruction that was in an epilogue, we search for the matching `EPILOGUE_BEG` note and move the note prior to the instruction. After doing so, we skip to the end of the basic block in which the note was located. By doing this, we ensure that all instructions that were in epilogues are "covered" by `EPILOGUE_BEG` notes and and therefore "covered" by the DWARF2 `remember_state` and `restore_state` byte codes described above.

3.4 Miscellanea

For our profiler to work properly, the application and all the libraries it requires must provide correct unwind information. It is preferable to have the compiler generate the necessary information, but we also plan to build a binary analysis tool to recover unwind information for legacy compilers that do not provide the necessary information. However, we have found that it is not simply sufficient to augment the compiler. While conducting the experiments described in Section 4, we found that assembly routines in `libc` were not properly annotated with unwind information. These routines required workarounds in the profiler similar to those already in place to detect samples taken within the trampoline. While these routines could have had their unwind information synthesized via binary analysis, it might be necessary for the assembly programmer to provide unwind information since binary analysis tools are not always successful at understanding the structure and semantics of machine code.

4 Experiments

Previous experiments [6] with `csprof` on Alpha/OSF1 demonstrated the viability of `csprof`'s approach for low-overhead profiling of unmodified, optimized binaries. However, the structure of the unwind information on x86-64 is very different from the unwind information on the Alpha. The Alpha uses a simpler, more compact unwind descriptor format that enables the unwinder to unwind using constant work per stack frame. We found, however, that the Compaq compilers did not always follow the published interface necessary for their unwind process to work correctly. Making `csprof` work correctly on a wide range of code required adding several work-arounds on the Alpha—in some cases as drastic as interpreting the instruction stream—to avoid problems caused by deviations from specifications. The workarounds for handling inaccurate unwind information contributed to profiler overhead, though overhead remained low despite them. Even though interpreting DWARF2 at run time would be more expensive than using Compaq's unwind information, we were optimistic that profiling overhead would be low on the x86-64 platform.

This section presents the results of experiments performed on the SPEC CPU2000 benchmarks [12] to gauge the impact of our changes on the size of compiled code as well as the effectiveness of `csprof` on the x86-64 platform. We ran our experiments on a 1.6GHz dual-processor Opteron with 8GB RAM using Gentoo Linux with the 2006.0 no-multilib profile.[4] We used GCC 4.1.0 with Gentoo's standard patches with options `-O3 -fomit-frame-pointer`

[4]We used Gentoo because it made it convenient to set up a `chroot` environment in which we had full control over how the system libraries were compiled. That the entire system was compiled with our modified GCC is a testament to its robustness.

`-funroll-all-loops` `-fno-tree-salias`. For unwinding call stacks within `csprof`, we used `libunwind` [10] version 0.98.5 with patches for x86-64 and an unwind cache for DWARF2.[5]

We encountered difficulties with several of the SPEC benchmarks. `255.vortex` would compile with both modified and unmodified versions of GCC 4.1.0, but would not run; we attribute this to some patch that Gentoo applies. `186.crafty` would not run consistently when compiled for profiling with `gprof` and we omitted it from Table 1. `csprof` was unable to profile `253.perlbmk` due to that benchmark's use of `longjmp`; `csprof`'s handling of `longjmp` on x86-64 is incomplete at present. Finally, `178.galgel` would not compile under the standard Gentoo compiler, which we again attribute to some patches Gentoo applies.

4.1 Profiling Overhead

This section reports measurements of `csprof`'s profiling overhead and compares it to that of `gprof`. We compiled each benchmark twice, with and without `-pg`.[6] To match `gprof`, `csprof`'s sampling frequency was set at 1000 samples/second. Results from the three runs are shown in Table 1. The runtimes in Table 1 are the average of five runs for each benchmark.

`csprof`'s overheads, shown in column three, are consistently lower than `gprof`'s overheads displayed in column two. Benchmarks such as `252.eon`, `253.perlbmk`, `168.wupwise`, and `177.mesa` have a high number of calls and a number of short procedures.[7] These sorts of programs demonstrate one of the primary problems with `gprof`: the overhead due

Integer programs

Benchmark	Runtime (seconds)	`gprof` overhead (percent)	`csprof` overhead (percent)
164.gzip	163	34	1.8
175.vpr	167	26	2.4
176.gcc	107	46	1.9
181.mcf	335	10	1.2
186.crafty	72	N/A	4.2
197.parser	281	50	1.8
252.eon	80	193	5.0
253.perlbmk	177	167	N/A
254.gap	128	174	1.5
255.vortex	N/A	N/A	N/A
256.bzip2	174	76	4.0
300.twolf	320	43	3.4
Average		82	2.7

Floating-point programs

Benchmark	Runtime (seconds)	`gprof` overhead (percent)	`csprof` overhead (percent)
168.wupwise	155	111	7.7
171.swim	260	15	3.1
172.mgrid	203	10	1.0
173.applu	256	25	1.6
177.mesa	124	74	1.6
179.art	210	3.8	1.4
183.equake	137	30	8.8
187.facerec	256	16	1.5
188.ammp	214	12	1.8
189.lucas	176	0	1.7
191.fma3d	264	28	1.1
200.sixtrack	235	1.7	0.9
301.apsi	259	16	3.5
Average		31	3.2

Table 1: Execution time overhead when profiling the SPEC CPU2000 benchmarks with `gprof` and `csprof`. A overhead of 100% indicates the monitored execution took twice as long.

[5]Available through a Mercurial repository at `http://www.serpentine.com/~arun/`

[6]Using the `-pg` flag required omitting `-fomit-frame-pointer`.

[7]We would expect `gprof`'s overhead on `255.vortex` to be high as well.

to instrumentation in prologues is unacceptably high in many cases. In contrast, `csprof` consistently has low overhead. Furthermore, `csprof`'s overhead can be reduced if necessary by lowering its sample frequency. The lone case where `gprof` has lower overhead than `csprof` is on `189.lucas`, which makes an extremely low number of calls. Even so, in this case `csprof`'s overhead is only 1.7%.

4.2 Space Overhead

`csprof`'s need for extra DWARF2 information for epilogues can increase the size of executables. Here we study this effect. For these tests, we compiled the CPU2000 benchmarks with Gentoo's vanilla GCC 4.1.0 package and our modified version of GCC 4.1.0 that records more complete unwind information. After doing so, we examined the size of the `.eh_frame` section, which contains the information for the DWARF2 unwinder. The results are shown in Table 2.

Table 2 shows that increases in the unwind information hover around 50%, with a few outliers such as `177.mesa` and `254.gap`. One might expect that the size of unwind information might double since we now record information about epilogues in addition to the information about prologues already recorded. We see less than a 2x space increase for two reasons. First, the encoding of epilogue unwind information using `DW_CFA_restore`, which specifies when a callee-saved register is restored, takes two bytes (opcode and register). This requires less space than its counterpart in prologues, `DW_CFA_offset`, which uses three bytes (opcode, offset, and register) to define a stack position for a register. Second, epilogue information occasionally fills space that would otherwise be filled by `DW_CFA_nops` inserted to satisfy alignment restrictions.

Integer programs

Benchmark	`.eh_frame` section size (bytes)	% increase
164.gzip	2604	51.3
175.vpr	6324	51.4
176.gcc	60812	62.6
181.mcf	956	45.2
186.crafty	4004	45.4
197.parser	12284	55.3
252.eon	34940	28.3
253.perlbmk	31724	56.2
254.gap	30140	78.1
255.vortex	32580	42.3
256.bzip2	2524	41.2
300.twolf	7732	52.9
Floating-point programs		
168.wupwise	980	35.9
171.swim	420	38.1
172.mgrid	756	48.7
173.applu	916	50.4
177.mesa	30804	83.1
179.art	1100	53.8
183.equake	1020	36.9
187.facerec	1604	49.4
188.ammp	7268	59.3
189.lucas	364	44.0
191.fma3d	17676	45.7
200.sixtrack	9900	57.9
301.apsi	4804	47.0

Table 2: Sizes in bytes of the `.eh_frame` section of applications when compiled with "vanilla" GCC 4.1.0 and the percentage increase with our modifications.

It is important to note that even though the relative increase in the size of the unwind information is large, this change was a small impact on the total size of the binary file on disk. Across all applications in the SPEC benchmark, our modifications increase the size of the binary file by less than two percent. We feel that this is a small cost to pay for the ability to effectively and informatively profile any fully optimized binary with low overhead. Furthermore, any application that might possibly need to unwind from within procedure epilogues (such as GDB) requires these modifications for correctness.

5 Related Work

Several call stack profilers are in wide use today. Apple's Shark [3] is a statistical call path profiler based on stack sampling. Like `csprof`, Shark provides full calling context for profiled costs. `csprof` goes one step further than Shark by recording return counts along edges, enabling more precise analysis of performance problems. Two well-known Linux call stack profilers, OProfile [8] and Sysprof [11], are system-wide profilers that require kernel-level support. Both do their call stack unwinding in the kernel and are limited to unwinding code compiled with frame pointers. This restriction almost certainly requires a recompile on x86-64, as the ABI for that platform does not require a frame pointer to be used. `csprof` has no such limitation.

Arnold and Sweeney [4] implemented a call stack profiler similar to `csprof` in Java. Since they controlled the run time environment and the compiler they implemented their sentinels by setting the low-order bit of the return address in every stack frame that their unwinder visited to indicate it had been seen. In subsequent unwinds only frames with their low-order bit cleared needed to be unwound. Their technique did not require any explicit compiler support, since the Java environment ignored the low-order bit of the return address when returning from a procedure. In Arnold and Sweeney's formulation, return counting could be done concurrently with the unwinding of the call stack during sampling—a "lazy" approach. Our technique gives us the freedom to choose between a lazy approach or an eager approach. We have chosen the eager approach because we already maintain a node into the CCT—to support efficient insertion of collected samples—and it is a simple matter to increment its return count in the trampoline.

6 A Call to Modernize `binutils`

We envision an overhaul of `binutils` shaped and informed by the needs of *performance tools*. We are *not* advocating a fundamental shift in the basic purpose of `binutils`; it currently is a collection of binary and performance tools. Rather we are advocating changes that would both improve the existing functionality and improve performance.

As a motivating example, consider the fact that we are not aware of a compiler that generates the complete unwind information required by `csprof`, even if the information is needed for correctness in other applications. While we would not object if other compilers generated such information—several non-GNU compilers generate `gprof` instrumentation—making `csprof` independent from compiler support and potential recompilation is an important goal. Therefore, we are interested in inferring unwind information using static binary analysis. Enhancing the `binutils` library in several ways would help us do this in a portable fashion.

6.1 Interface and Functionality

First, to infer what instructions affect the stack frame and register state, the `binutils` API must offer improved support for examining a binary's instruction stream and determining where procedures begin and end. Even though `binutils` supports the disassembly of binaries on a wide range of platforms, the interface it exposes is specifically designed for printing disassembled instructions, not for examining or querying properties about all of the instructions in a procedure. A more general interface would not negatively affect the operation of `binutils` consumers such as `objdump` or GDB and could be exploited by a wide range of binary analysis tools.

Another example that motivates the examination of binaries is the `bloop` binary analysis tool, which is part of our HPCToolkit [9] performance analysis tools. `bloop` presently uses a modified version of `binutils` that we locally maintain. Given an executable, `bloop` constructs the control flow graph (CFG) for each procedure, performs interval analysis to recover loop nests, and consults the executable's line maps to map VMAs to source line numbers. `bloop` then correlates the recovered program structure information with profile data to compute loop-based performance metrics in addition to the traditional procedure and line based metrics.[8] To reconstruct the CFG, `bloop` must 1) classify instructions (conditional branch, unconditional jump, return, non-control-flow) and 2) compute the target address of PC-relative conditional branches. Since we knew that `objdump`'s disassembler regurgitated nearly all of this information, we wondered if `binutils`' opcode library could help us. However, because `binutils` does not make this information accessible to its clients, we nearly abandoned it after being initially discouraged by its print-biased interface. However, after a clever suggestion from a former `binutils` maintainer, we were able to extract the information we needed with a small amount of precise surgery. The small modifications we made to the interface and to the relevant decoders made a significant difference in `binutils`' utility. We understand that `binutils` was designed to meet a specific need; our argument is that a careful and modest redesign would continue to meet the needs of its current clients while providing the support needed for sophisticated binary analysis tools.

[8]By performing loop analysis within a binary analyzer instead of a source-level tool, `bloop` is able to analyze binaries from multi-language code bases (e.g. Fortran95, C, C++) without multiple front-ends. More interestingly, with careful use of accurate debugging information, a binary analyzer provides information about the actual *optimized* code, not just the source code, enabling us to understand the effects due to loop transformations, software pipelining and loop fusion.

6.2 Performance Issues

A second major concern with `binutils` is algorithmic efficiency. Since a typical binary analyzer must examine every instruction in every procedure of a load module, algorithmic complexity is an important consideration. For example, consider our proposed tool for synthesizing unwind information from an executable. If `csprof` dynamically invoked such a tool in response to a runtime call to `dlopen`, the tool must run very quickly. While performing a linear search through the line table to map an instruction address back to a source line might be reasonable for a client like GDB when execution stops at a breakpoint, such an approach would be unsuitable for a tool that uses this interface to map each instruction in a large executable back to its source line. These are not theoretical concerns since executables can easily have hundreds of thousands of instructions.

To reduce the execution time of `bloop`, we had to replace `binutils`'s linear searches through the DWARF2 and ECOFF line tables with binary searches. Since `bloop` maps every instruction back to its source line, linear search of the line table caused execution time to grow quadratically with procedure size. For a similar reason, we added a one-element cache to the ELF function name lookup since sorting the symbols was a difficult solution. Attention to algorithmic efficiency should also benefit current clients of `binutils`.

6.3 `binutils` vs. Custom Solutions

Performance analysis tools require access to a richer set of the symbolic and system information contained in typical binaries than currently exposed by `binutils`. For example, `bloop`'s analysis is complicated by nested procedures (e.g. Fortran 90) and by multiply instantiated statement instances (e.g. inlining). Since such information can be useful for debuggers, DWARF[9]—the nearly universal standard in the UN*X world—contains constructs for representing this information. Since most of this information was already read by the `binutils`'s DWARF reader, we wrote our own call-back routines to expose it in the `binutils` interface. In addition to symbolic information, binary analysis tools require access to the binary's system information, commonly represented as ELF in the UN*X world. We continue to need an easier way to access a binary's list of dependencies (the information retrieved by `ldd`) and to find the begin addresses of the segments that are mapped to memory during run time.

The references to DWARF and ELF raise the issue of portability, one of our initial goals for `csprof`. After all, the *raison d'etre* for `binutils` is to provide a common interface for a multitude of different ABIs. It also revisits the question of scope: what is the purpose of `binutils`? Why not use `libelf` and `libdwarf` to access specific ELF and DWARF information? Other performance tools authors have abandoned `binutils` in favor of creating their own binary interfaces for reasons related to the issues we have raised. We think, however, that a judicious redesign of `binutils` can reasonably accommodate the competing demands of, on one hand, generality and portability and on the other, improved and efficient support for the specialized and richer set of information encoded in ELF and DWARF.[10] Moreover, we argue that this is desirable because ELF and DWARF are such widely used standards, in fact *the* standard on Linux/GNU systems. A modernized and portable `binutils` that provides greater access to the instruction stream, emphasizes efficiency, and exposes more of the symbolic and system information (with a particular bias towards DWARF and ELF), would both improve `binutils` and provide an excellent platform for the development of a first-class suite of performance tools.

7 Conclusions and Future Work

We have presented `csprof`, a low-overhead call stack profiler and the modifications necessary to make it work on GCC. Experiments on the x86-64 platform have confirmed our initial results from experiments on the Alpha platform and have given us confidence that `csprof`'s approach offers a portable method for low-overhead context-sensitive profiling. Our methods are easily enabled by modest compiler support; however, to our knowledge no other compiler generates the necessary DWARF2 byte codes for `csprof` to unwind fully optimized code at arbitrary points, which is necessary with an asynchronous sample source. Binary analysis can provide the necessary information, but is not foolproof. Therefore, we would very much like our modifications to GCC to be integrated into the mainline compiler as both sup-

[9]We use 'DWARF' to include both DWARF2 and DWARF3.

[10]Note that we are not addressing the issue of binary modification. While this is an interesting area, support for these techniques would genuinely be a new concern for `binutils`.

port for our methods and to motivate other compiler groups to provide similar DWARF2 information.

Distributed in the `binutils` package, `gprof` has for many years offered portable, context-sensitive profiling for performance analysis and naturally complements GCC. To better support profiling of fully optimized modern object-oriented applications, we have designed `csprof` to deliver low-overhead profiles with full calling context for costs. Just as `gprof` required compiler support to make profiling easy to use, we have extended GCC to generate the unwind information required by `csprof`. We believe `csprof` should be a key component of a modernized `binutils`.

However, there remains work to be done before our work could be considered ready for GCC mainline. As Andrew Haley pointed out on the `gcc-patches` list,[11] we would need to ensure that our modifications worked with the x86 back end as well. We are in the process of satisfying this requirement, as our changes cause issues while performing the register-to-stack conversion pass necessary for the x86. In addition, our modifications prevent GCC's code reordering from working, as generating the DWARF2 `advance_offset` byte code requires taking the difference of two assembly code labels—an operation that fails when those labels reside in different sections. We are investigating options for alternate approaches for the necessary byte code generation. Finally, Haley also noted that with our patches, only the x86 and x86-64 back ends are capable of generating this extra information for unwinding. He felt that this sort of information should be generated consistently across all of GCC's back ends, as appropriate. While we agree with this point in principle, we feel that integrating our work is merely an enhancement to the x86 and x86-64 back ends and the necessary work for other back ends can be done as time permits.

[11]`http://gcc.gnu.org/ml/gcc-patches/2006-03/msg00426.html`

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, New York, NY, USA, 1997. ACM Press.

[2] 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? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[3] Apple Computer. Shark. `http://developer.apple.com/performance/`. 14 April 2006.

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

[5] C. de Dinechin. C++ exception handling. *IEEE Concurrency*, 8(4):72–79, 2000.

[6] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN*

Symposium on Compiler Construction, pages 120–126, New York, NY, USA, 1982. ACM Press.

[8] John Levon et al. OProfile. `http://oprofile.sf.net/`. 14 April 2006.

[9] 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.

[10] D. Mosberger-Tang. `libunwind`. `http://www.hpl.hp.com/research/linux/libunwind/`. 14 April 2006.

[11] S. Sandmann. Sysprof. `http://www.daimi.au.dk/~sandmann/sysprof/`. 14 April 2006.

[12] SPEC Corporation. SPEC CPU2000 benchmark suite. `http://www.spec.org/cpu2000/`. 29 April 2005.

[13] UNIX International. DWARF debugging information format. `http://www.eagercon.com/dwarf/dwarf-2.0.0.pdf`. 29 April 2005.