Binary Analysis for Measurement and Attribution of Program Performance

Nathan R. Tallent John M. Mellor-Crummey Michael W. Fagan
Rice University
{tallent,johnmc,mfagan}@rice.edu

Abstract

Modern programs frequently employ sophisticated modular designs. As a result, performance problems cannot be identified from costs attributed to routines in isolation; understanding code performance requires information about a routine's calling context. Existing performance tools fall short in this respect. Prior strategies for attributing context-sensitive performance at the source level either compromise measurement accuracy, remain too close to the binary, or require custom compilers. To understand the performance of fully optimized modular code, we developed two novel binary analysis techniques: 1) *on-the-fly* analysis of optimized machine code to enable minimally intrusive and accurate attribution of costs to dynamic calling contexts; and 2) post-mortem analysis of optimized machine code and its debugging sections to recover its program structure and reconstruct a mapping back to its source code. By combining the recovered static program structure with dynamic calling context information, we can accurately attribute performance metrics to calling contexts, procedures, loops, and inlined instances of procedures. We demonstrate that the fusion of this information provides unique insight into the performance of complex modular codes. This work is implemented in the HPCTOOLKIT[1] performance tools.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes.

General Terms Performance, Measurement, Algorithms.

Keywords Binary analysis, Call path profiling, Static analysis, Performance tools, HPCTOOLKIT.

1. Introduction

Modern programs frequently employ sophisticated modular designs that exploit object-oriented abstractions and generics. Composition of C++ algorithm and data structure templates typically yields loop nests spread across multiple levels of routines. To improve the performance of such codes, compilers inline routines and optimize loops. However, careful hand-tuning is often necessary to obtain top performance. To support tuning of such code, performance analysis tools must pinpoint context-sensitive inefficiencies in fully optimized applications.

Several contemporary performance tools measure and attribute execution costs to calling context in some form. However, when applied to fully optimized applications, existing tools fall short for two reasons. First, current calling context measurement techniques are unacceptable because they either significantly perturb program optimization and execution with instrumentation, or rely on compiler-based information that is sometimes inaccurate or unavailable, which causes failures while gathering certain calling contexts. Second, by inlining procedures and transforming loops, optimizing compilers introduce a significant semantic gap between the binary and source code. Thus, prior strategies for attributing context-sensitive performance at the source level either compromise measurement accuracy or remain too close to the object code.

To clarify, we consider the capabilities of some popular tools using three related categories: calling context representation, measurement technique and attribution technique.

Calling context representation. Performance tools typically attribute performance metrics to calling context using a call graph or call path profile. Two widely-used tools that collect call graph profiles are `gprof` [11] and Intel's VTune [15]. A *call graph profile* consists of a node for each procedure and a set of directed edges between nodes. An edge exists from node p to node q if p calls q. To represent performance measurements, edges and nodes are weighted with metrics. Call graph profiles are often insufficient for modular applications because a procedure p that appears on multiple distinct paths is represented with one node, resulting in shared paths and cycles. Consequently, with a call graph profile it is in general not possible to assign costs to p's full calling context, or even to long portions of it. To remove this imprecision, a *call path profile* [12] represents the full calling context of p as the path of calls from the program's entry point to p. Call path profiling is necessary to fully understand the performance of modular codes.

Measurement technique. There are two basic approaches for obtaining calling context profiles: instrumentation and statistical sampling. Instrumentation-based tools use one of three principal instrumentation techniques. Tools such as Tau [26] use *source code instrumentation* to insert special profiling code into the source program before compilation. In contrast, VTune [15] uses *static binary instrumentation* to augment application binaries with profiling code. (`gprof`'s [11] instrumentation, though traditionally inserted by a compiler, is effectively in this category.) The third technique is *dynamic binary instrumentation*.

While source-level instrumentors collect measurements that are easily mapped to source code, their instrumentation can interfere with compiler optimizations such as inlining and loop transformations. As a result, measurements using source-level instrumentation may not accurately reflect the performance of fully optimized

[1] HPCTOOLKIT is an open-source suite of performance tools available from `http://hpctoolkit.org`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06. . . $5.00

code [28]. Binary instrumentors may also compromise optimization. For example, in some compilers `gprof`-instrumented code cannot be fully optimized.

An important problem with both source and static binary instrumentation is that they require recompilation or binary rewriting of a program and all its libraries. This requirement poses a significant inconvenience for large, complex applications. More critically, the need to see the whole program before run time can lead to 'blind spots,' *i.e.*, portions of the execution that are systematically excluded from measurement. For instance, source instrumentation fails to measure any portion of the application for which source code is unavailable; this frequently includes critical system, math and communication libraries. For Fortran programs, this approach can also fail to associate costs with intrinsic functions or compiler-inserted array copies. Static binary instrumentation is unable to cope with shared libraries dynamically loaded during execution.

The third approach, *dynamic binary instrumentation*, supports fully optimized binaries and avoids blind spots by inserting instrumentation in the executing application [4]. Intel's recently-released Performance Tuning Utility (PTU) [14], includes a call graph profiler that adopts this approach by using Pin [18]. However, dynamic instrumentation remains susceptible to systematic measurement error because of instrumentation overhead.

Indeed, all three instrumentation approaches suffer in two distinct ways from overhead. First, instrumentation dilates total execution time, sometimes enough to preclude analysis of large production runs or force users to *a priori* introduce blind spots via selective instrumentation. For example, because of an average slowdown factor of 8, VTune *requires* users to limit measurement to so-called 'modules of interest' [15]. Moreover, overhead is even more acute if loops are instrumented. A recent Pin-based 'loop profiler' incurred an average slowdown factor of 22 [22]. Second, instrumentation dilates the total measured cost of each procedure, disproportionately inflating costs attributed to small procedures and thereby introducing a systematic measurement error.

The alternative to instrumentation is *statistical sampling*. Since sampling periods can easily be adjusted (even dynamically), this approach naturally permits low, controllable overhead. Sampling-based call path profilers, such as the one with Intel's PTU [14], use call stack unwinding to gather calling contexts. Stack unwinding requires either the presence of frame pointers or correct and complete unwind information for *every* point in an executable because an asynchronous sample event may occur anywhere. However, fully optimized code often omits frame pointers. Moreover, unwind information is often incomplete (for epilogues), missing (for hand-coded assembly or partially stripped libraries) or simply erroneous (optimizers often fail to update unwind information as they transform the code). In particular, optimized math and communication libraries frequently apply every 'trick in the book' to critical procedures (*e.g.*, hot-cold path splitting [6])— just those procedures that are likely to be near the innermost frame of an unwind.

Attribution technique. By inlining procedures and transforming loops, optimizing compilers introduce a semantic gap between the object and source code, making it difficult to reconcile binary-level measurements with source-level constructs. Compiler transformations such as inlining and tail call optimization cause call paths during execution to differ from source-level call paths. After compilers inline procedures and apply loop transformations, execution-level performance data does not correlate well with source code. Since application developers wish to understand performance at the source code level, it is necessary for tools to collect measurements on fully optimized binaries and then translate those measurements into source-level insight. Since loops are critical to performance, but are often dynamically nested across procedure calls, it is important to understand loops in their calling context.

Much prior work on loop attribution either compromises measurement accuracy by relying on instrumentation [22, 26] or is based on context-less measurement [19]. A few sampling-based call path profilers [2, 14, 22] identify loops, but at the binary level. Moseley *et al.* [22] describe a sampling-based profiler (relying on unwind information) that additionally constructs a dynamic loop/call graph by placing loops within a call graph. However, by not accounting for loop or procedure transformations, this tool attributes performance only to binary-level loops and procedures. Also, by using a dynamic loop/call *graph*, it is not possible to understand the performance of procedures and loops in their full calling context.

To understand the performance of modular programs, as part of the HPCTOOLKIT performance tools we built `hpcrun`, a call path profiler that measures and attributes execution costs of unmodified, fully optimized executables to their full calling context, as well as loops and inlined code. Achieving this result required novel solutions to three problems:

1. To measure dynamic calling contexts, we developed a context-free on-line binary analysis for locating procedure bounds and computing unwind information. We show its effectiveness on applications in the SPEC CPU2006 suite compiled with Intel, Portland Group and PathScale compilers using peak optimization.
2. To attribute performance to user-level source code, we developed a novel post-mortem analysis of the optimized object code and its debugging sections to recover its program structure and reconstruct a mapping back to its source code. The ability to expose inlined code and its relation to source-level loop nests without a special-purpose compiler and without any additional measurement overhead is unique.
3. To compellingly present performance data, we combine (post-mortem) the recovered static program structure with dynamic call paths to expose inlined frames and loop nests. No other sampling-based tool attributes the performance of *transformed loops* in the *full calling context* of *transformed routines* for *fully optimized binaries* to source code.

In this paper, we describe our solutions to these problems. The major benefit of our approach is that `hpcrun` is minimally invasive, yet accurately attributes performance to both static and dynamic context, providing unique insight into program performance.

Our results are summarized by Figure 1. As shown in Figure 1(a), let $p \to q \to r \to s$ be a user-level call chain of four procedures. Procedure p contains a call site c_p (that calls q) embedded in loop l_p; procedures q and r contain analogous call sites. Assume that a compiler inlines call site c_q so that code for procedure r appears within loop l_q. Consequently, at run time c_q is not executed and therefore a procedure frame for r is absent. Using call stack unwinding and line map information recorded by compilers yields the reconstruction of context shown in Figure 1(c). By combining dynamic context obtained by call stack unwinding with static information about inlined code and loops gleaned using binary analysis, our tools obtain the reconstruction shown in Figure 1(b). Specifically, our tools 1) identify that c_p and c_r are located within loops; 2) detect the inlining; and 3) nest c_r within both its original procedure context r and its new host procedure q. Most importantly, reconstructed procedures, loops and inlined frames can be treated as 'first-class' entities for the purpose of assigning performance metrics.

The rest of the paper is as follows. Section 2 describes our use of binary analysis to support call path profiling of optimized code and evaluates its effectiveness (contribution 1). Section 3 describes

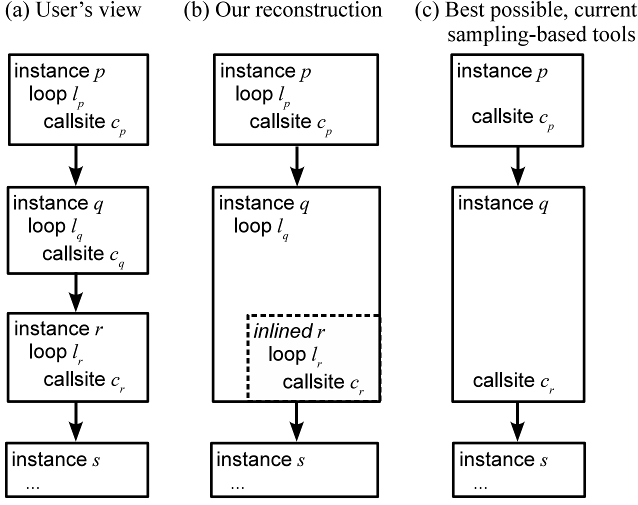


Figure 1. Correlating call paths with program structure.

our binary analysis to support accurate correlation of performance measurements to optimized code (contribution 2). Section 4 highlights the rich performance data we obtain by fusing dynamic call paths and static structure (contribution 3). Finally, Section 5 discusses related work; and Section 6 summarizes our conclusions.

2. Binary Analysis for Call Path Profiling

Sampling-based call path profilers use call stack unwinding to gather calling contexts. For such profilers to be accurate, they must be able to unwind the call stack at *any* point in a program's execution. A stack unwind, which forms the calling context for a sample point, is represented by the program counter for the innermost procedure frame and a list of return addresses — one for each of the other active procedure frames. Successfully unwinding the call stack requires determining the return address for each frame and moving up the call chain to the frame's parent. Obtaining the return address for a procedure frame without a frame pointer is non-trivial since the procedure's frame can dynamically grow (as space is reserved for the caller's registers and local variables, or supplemented with calls to `alloca`) and shrink (as space for the aforementioned purposes is deallocated) as the procedure executes. If the return address is kept in the stack (as is typical for non-leaf procedures), the offset from the stack pointer at which the return address may be obtained often changes as a procedure executes.

Finding the return address for a procedure frame is simple with correct and complete compiler-generated unwind information [10]. Unfortunately, compilers routinely omit unwind information for procedure epilogues because it is not needed for exception handling. However, even if compilers generate complete unwind information, fully optimized applications often link with vendor libraries (*e.g.*, math or OpenMP) that have incomplete unwind tables due to hand-coded assembly or partial stripping. Since codes may spend a significant fraction of time in procedures that lack proper unwind information,[2] dropping or mis-attributing samples that occur in such procedures could produce serious measurement error.

To enable accurate unwinding of *all* code, even code lacking compiler-based unwind information, we developed two binary analyzers — one to determine where a procedure begins and ends in partially-stripped code, and a second to compute how to unwind to

Algorithm 1: High-level sketch of using on-the-fly binary analysis to support call stack unwinding of optimized code.

Input: $\mathcal{B}$, procedure bounds for each load module
Input: $\mathcal{U}$, unwind recipes for procedure intervals (splay tree)

let $\mathcal{F} = \langle PC, FP, SP \rangle$ be the frame of the sample point (consisting of program counter, frame and stack pointer)
while *$\mathcal{F}$ is not the outermost frame* **do**
  if *$\mathcal{U}$ has no unwind recipe for PC* **then**
    let μ be the load module containing PC
    if *$\mathcal{B}$ has no bounds for μ* **then**
      Compute bounds for all procedures in μ
    let π be the procedure (from $\mathcal{B}$) with bounds β containing PC
    Scan the object code of π, 1) tracking the locations of its caller's program counter, frame and stack pointer; and 2) creating an unwind recipe for each distinct interval
  let υ be the unwind recipe (from $\mathcal{U}$) for PC
  let $\mathcal{F}' = \langle PC', FP', SP' \rangle$ be the caller's frame, computed using υ
  $\mathcal{F} \Leftarrow \mathcal{F}'$

a caller's frame from any address within a procedure. At any instant, a frame's return address (which also serves as the program counter for the calling frame) may be located either 1) in a register, 2) in a location relative to the stack pointer, or 3) in a location relative to the frame pointer (which the frame must have initialized before using). The value of the frame pointer for a caller's frame may be found similarly. To recover the program counter, stack pointer and frame pointer values for a caller's frame, we compute a sequence of *unwind recipes* for a procedure. Each unwind recipe corresponds to an interval of code that ends in a *frame-relevant instruction*. A frame-relevant instruction is one that changes the machine state (*e.g.*, by moving the stack pointer, saving the frame pointer value inherited from the caller, or initializing the frame pointer for the current frame) in such a way that a different unwind recipe is needed for instructions that follow.

Although procedure bounds and unwind recipes could be computed off-line, we perform both analyses on demand at run time. We perform binary analysis on each load module to recover the bounds of all of its procedures. This analysis is triggered at program launch for the executable and all shared libraries loaded at launch and whenever a new shared library is loaded with `dlopen`. The computed procedure-bounds information for a module is cached in a table that is queried using binary search. We perform binary analysis to compute unwind intervals for a procedure lazily — the first time that the procedure appears on the call stack when a sample event occurs. This approach elegantly handles dynamically loaded shared libraries and avoids wasting space and time computing unwind recipes for procedures that may never be used. To support fast queries, we memoize unwind recipes in a splay tree indexed by intervals of code addresses. Algorithm 1 shows a high-level overview of the process of performing on-the-fly binary analysis to support call path profiling. Because dynamic analysis must be efficient, we prefer fast linear-time heuristics that may occasionally fail over slower fully general methods.[3] (An evaluation of our approach in Section 2.3 shows that our methods almost never fail in practice.) In the next two sections, we describe how we infer procedure bounds and compute unwind recipes.

[2] For example, the S3D turbulent combustion code described in Section 4.2 spends nearly 20% of its total execution time in the math library's exponentiation routine as it computes reaction rates.

[3] For example, Rosenblum *et al.* [24] developed an off-line analyzer to recover procedure bounds in fully stripped code. However, the focus of this work was on thorough analysis for security.

2.1 Inferring Procedure Bounds

To compute unwind recipes for a procedure based on its instruction sequence, one must know the procedure's *bounds*, namely where the procedure begins and ends. In many cases, complete information about procedure bounds is not readily available. For instance, stripped shared libraries have only a dynamic symbol table that contains only information about global procedure symbols; all information about local symbols is missing. Often, libraries are partially stripped. For instance, the OpenMP run time library for version 3.1 of PathScale's x86-64 compiler only has symbol information for OpenMP API procedures; all information about other procedures is missing. For this reason, inferring procedure bounds for stripped or partially stripped code is an important precursor to computing unwind intervals.

Our approach for inferring procedure bounds is based on the following observations.

- *We expect each load module to provide information about at least some procedure entry points.*

  Performance analysis of a stripped executable is typically unproductive. Interpreting measurement results is difficult without procedure names. For this reason, entry points for user procedures will generally be available for an executable. Dynamically-linked shared libraries have (at a minimum) procedure entry points for externally-visible library procedures.

- *We must perform procedure discovery on all load modules.*

  Partially-stripped libraries are not uncommon. There is no *a priori* way to distinguish between a partially-stripped load module and one that has full symbol information. We have also encountered (non-stripped) executables that lack information about some procedures. For instance, the SPEC benchmark xalancbmk, when compiled with the PathScale C++ compiler (version 3.1, using `-O3`) contains small anonymous procedures.

- *Having the proper address for a procedure start is more important than having the proper address for a procedure end.*

  For a procedure with the interval $[s, e)$, incorrectly inferring the procedure end at address $e' > e$ will not change the unwind recipes that we compute for the interval $[s, e)$.

- *We assume all procedures are contiguous*

  In other words, we assume a single procedure is *not* divided into disjoint code segments. For the most part, this assumption holds. We have, however, encountered compilers that employ hot-cold optimization [6]. This optimization sometimes splits the procedure into disjoint segments. Furthermore, an unrelated procedure may be placed between the disparate parts of the hot-cold-optimized procedure. Our treatment of a divided procedure is to treat each part as a separate procedure. Our treatment simplifies procedure discovery, but requires additional consideration when determining the unwind recipe for the various segments of a divided procedure. See Section 2.2 for more information.

- *Not all false positives are equally problematic.*

  We classify false procedures starts into two categories: *malignant* and *benign*. If we infer a false procedure start in a gap between two real procedures that contains data (*e.g.*, a jump table for a switch statement), this will not affect the bounds of any real procedures for which we need to compute unwind intervals. For this reason, we call such a false procedure start benign. On the other hand, if we infer a false procedure start s' in the middle of a real procedure ranging from $[s, e)$, this may cause us to compute incorrect unwind information for the interval $[s', e)$. We call such a false procedure start malignant.

2.1.1 Approach

We take an aggressive approach to procedure discovery. Without evidence to the contrary, we assume that the instruction following an unconditional jump or a return is the start of a new procedure. In optimized code, we have also seen procedures that end with a call to a procedure that doesn't return (*e.g.*, `exit` or `abort`). To handle this case, we infer a function start after a call if we immediately encounter code that is obviously a function prologue. We use the following collection of heuristics to avoid inferring a procedure start within a procedure (a malignant false positive).

- We call the interval between a conditional branch at an address a and its target at address t a *protected interval*. No procedure start will be inferred in a protected interval. If $a < t$, this yields a protected interval $[a, t')$, where t' is the end of the instruction at address t; otherwise, this yields a protected interval $[t, a')$, where a' is the end of the instruction at address a. (Conditional jumps are almost always within procedures. While we have found one or two conditional forward branches used as tail calls in `libc`, other heuristics prevent us from missing procedure starts in this rare case.)
- A backward unconditional jump at address a into a protected interval that extends from $[s, e)$ extends the protected interval to cover the range $[s, a')$, where a' is the end of the instruction at address a. (Such jumps often arise at the end of 'cold path' prefetching code that has been outlined from loops and deposited after what would have been the end of the procedure.)
- Moving the stack pointer upward at address a in a procedure prologue (to allocate stack space for local variables) must be followed by a compensating adjustment of the stack pointer in each of the procedure's n epilogues, at addresses $e_1, \ldots, e_n$. Let e_n be the epilogue with the largest address. We treat the interval $[a, e_n')$ as a protected.
- Let the interval between initializing the frame pointer register with the value of the stack pointer and restoring the value of the frame pointer be a protected interval. Similarly, let the interval between a 'store' and 'load' of the frame pointer be a protected interval.
- A global symbol in the symbol table or the dynamic symbol table is always considered a procedure start, even if it lies within a protected interval. In contrast, a local symbol only considered a procedure start if it does not fall within a protected interval.

2.2 Computing Unwind Recipes

Because dynamic analysis must be efficient, we prefer fast linear-time heuristics that are typically accurate over slower fully general methods. Experiments described in Section 2.3 show that our approach is nearly perfect in practice. Although we initially developed our strategy for computing unwind recipes for x86-64 binaries, the general approach is architecture independent. We recently adapted it to compute unwind recipes for MIPS and PowerPC binaries to support call path profiling on SiCortex clusters and Blue Gene/P, respectively.

Our binary analyzer creates an unwind recipe for each distinct interval within a procedure. An interval is of the form $[s, e)$ and its unwind recipe describes where to find the caller's program counter, frame pointer (FP) register value, and stack pointer (SP). For example, the caller's program counter (the current frame's return address) can be in a register, at an offset relative to SP or at an offset relative to FP; the value of the caller's FP register, which may or may not be used by the caller as a frame pointer, is analogous.

The initial interval begins with (and includes) the first instruction. The recipe for this interval describes the frame's state immediately after a call. For example, on x86-64, a procedure frame begins

with its return address on the top of stack, the caller's value of FP in register FP, and the caller's value of SP at SP-8, just below the return address. In contrast, on MIPS, the return address is in register RA and the caller's value of FP and SP are in registers FP and SP, respectively.

The analyzer then computes unwind recipes for each interval in the procedure by determining where each interval ends. (Intervals are contiguous and cannot overlap.) To do this, it performs a linear scan of each instruction in the procedure. For each instruction, the analyzer determines whether that instruction affects the frame. (For x86-64, where instruction decoding is challenging, we use Intel's XED tool [5].) If so, the analyzer ends the current interval and creates a new interval at the next instruction. The unwind recipe for the new interval is typically created by applying the instruction's effects to the previous interval's recipe. An interval ends when an instruction:

1. modifies the stack pointer (pushing registers on the stack, subtracting a fixed offset from SP to reserve space for a procedure's local variables, subtracting a variable offset from SP to support `alloca`, restoring SP with a frame pointer from FP, popping a saved register),
2. assigns the value of SP to FP to set up a frame pointer,
3. jumps using a constant displacement to an address outside the bounds of the current procedure (performing a tail call),
4. jumps to an address in a register when SP points to the return address,
5. returns to the caller,
6. stores the caller's FP value to an address in the stack, or
7. restores the caller's FP value from a location in the stack.

There are several subtleties to the process sketched above: following a return or a tail call (items 4 and 5 above), a new interval begins. What recipe should the new interval have? We initialize the interval following a tail call or a return with the recipe for the interval that we identify as the *canonical frame*. We use the following heuristic to determine the canonical frame $\mathbf{C}$. If a frame pointer relative (FP) interval was found in the procedure (FP was saved to the stack and later initialized to SP), let $\mathbf{C}$ be the first FP interval. Otherwise, we continue to advance $\mathbf{C}$ along the chain of intervals while the frame size (the offset to the return address from the SP) is non-decreasing, and the interval does not contain a branch, jump, or call. We use such an interval as a signal that the prologue is complete and the current frame is the canonical frame. In addition, whenever a return instruction is encountered during instruction stream processing, we check to make sure that the interval has the expected state: *e.g.*, for x86-64, the return address should be on top of the stack, and the FP should have been restored. If the interval for the return instruction is not in the expected state, then the interval that was most recently initialized from the canonical frame is at fault. When a return instruction interval anomaly is detected, we adjust all of the intervals from the interval reaching the return back to the interval that was most recently initialized from the canonical frame.

To handle procedures that have been split via hot-cold optimization, we check the end of the current procedure p for a pattern that indicates that p is not an independent procedure, but rather part of another one. The pattern has two parts:

1. p ends with an unconditional branch to an address a that is in the interior of another procedure q.
2. The instruction preceding a is conditional branch to the beginning of p.

When the hot-cold pattern is detected, all intervals in p are adjusted according to the interval computed for a.

Integer programs

Benchmark	Overhead (percent) hpc run	PTU-smpl	PTU-Pin	Unwind Failures hpc run	PTU-smpl Intel	Others
perlbench	1.3	0.9	1043.3	0.0	4.5%	87.5%
bzip2	2.9	0.9	197.1	0.0	0.8%	52.2%
gcc	3.2	1.3	300.9	15.1	4.5%	70.7%
mcf	1.3	2.6	8.5	0.0	0.1%	60.4%
gobmk	1.7	1.3	481.3	0.1	2.4%	71.6%
hmmer	0.4	1.0	36.4	0.0	0.1%	74.4%
sjeng	0.3	1.6	694.4	0.0	19.2%	100.0%
libquantum	-0.2	-0.2	16.3	0.0	0.1%	99.9%
h264ref	0.1	0.0	784.2	0.6	21.9%	69.7%
omnetpp	1.6	1.7	701.2	0.0	1.4%	49.4%
astar	1.6	1.7	184.1	0.0	0.5%	57.6%
xalancbmk	9.5	10.8	732.0	0.0	1.0%	0.4%
Average*	2.0	1.9	431.6	1.3	4.7%	66.1%
Std. Dev.	2.6	2.8	353.4	4.3	7.6%	26.6%

Floating-point programs

Benchmark	hpc run	PTU-smpl	PTU-Pin	hpc run	PTU-smpl Intel	Others
bwaves	1.7	1.9	9.9	0.0	0.0%	66.6%
gamess	0.8	0.1	†	0.0	0.3%	99.7%
milc	0.6	0.4	61.0	0.0	0.0%	99.9%
zeusmp	2.1	2.0	†	0.0	0.0%	99.7%
gromacs	0.6	0.4	57.3	0.0	0.1%	100.0%
cactusADM	1.6	1.5	6.7	0.0	0.0%	100.0%
leslie3d	2.0	1.7	2.5	0.0	0.0%	93.5%
namd	0.2	1.5	5.1	0.0	0.0%	42.0%
dealII	0.5	0.7	1746.4	0.0	2.7%	83.8%
soplex	1.6	1.8	19.3	0.0	2.0%	54.3%
povray	0.1	0.3	1732.8	0.0	6.5%	49.8%
calculix	-0.5	0.9	62.5	0.0	0.2%	99.5%
GemsFDTD	-0.8	-1.2	45.3	0.0	0.1%	74.9%
tonto	0.3	1.3	287.4	0.0	11.1%	98.0%
lbm	0.9	1.2	10.2	0.0	0.0%	13.5%
wrf	3.0	1.5	59.5	0.5	0.0%	98.2%
sphinx3	0.4	2.4	84.7	0.0	1.9%	48.0%
Average*	0.9	1.1	279.4	0.0	1.5%	77.7%
Std. Dev.	1.0	0.9	566.0	0.1	3.0%	27.1%

* Neither the arithmetic nor geometric mean summarizes these values well.
† PTU-Pin failed to execute any version of these benchmarks.

Table 1. Comparing `hpcrun` and PTU on SPEC CPU2006.

In the linear scan between the start and end address of a procedure, the analyzer may encounter embedded data such as jump tables. This may cause decoding to fail or lead to corrupt intervals that would leave us unable to unwind. Although such corrupt intervals could cause unwind failures (we note such failures in a log file), we have not found them to be a problem in practice.

2.3 Evaluation

To evaluate the efficiency and effectiveness of our binary analyses for unwinding against contemporary tools, we compared `hpcrun` with two of the tools from Intel's Performance Tuning Utility (PTU) [14] — PTU's call stack sampling profiler (PTU-smpl) and PTU's Pin-based call graph profiler (PTU-Pin) — using the SPEC CPU2006 benchmarks [27]. Since PTU is designed for Intel architectures, this evaluation focuses on analysis of x86-64 binaries. We compiled two versions of each benchmark, distinguished by 'base' or 'peak' optimization, using the Intel 10.1 (20080312), PathScale 3.1 and Portland Group (PGI) 7.1-6 compilers; this resulted in six versions of each benchmark. We used the following 'base' and 'peak' optimization flags: for Intel, `-O3` and `-fast` (but with static linking disabled); for PathScale, `-O3` and `-Ofast`; for PGI, `-fast -Mipa=fast,inline`. To permit high-throughput testing, experiments were performed on a cluster where each node is a dual-socket Intel Xeon Harpertown (E5440) with 16 GB memory running Red Hat Enterprise Linux 5.2. Table 1 summarizes our results.

2.3.1 Efficiency

The first multi-column of Table 1 compares the average overhead of `hpcrun` with PTU-smpl and PTU-Pin. We first observe that despite PTU-Pin's sophistication, dynamic binary instrumentation is not an acceptable measurement technique for two reasons. First, compared to a worst case sampling overhead of about 10% (average of 1-2%), instrumentation can introduce slowdown *factors* of 10-18. Second, the drastic variation in overheads strongly suggests that Pin's instrumentation dilates the execution of small procedures and introduces systematic distortion. Because of the extremely long run times and the clear advantage of sampling, we chose not to collect PTU-Pin results on executables generated by non-Intel compilers, assuming that an Intel tool used with an Intel-generated executable represents a best-case usage.

Both `hpcrun`'s and PTU-smpl's results are averaged over all six versions of the benchmarks; each tool used a 5 ms sampling period, yielding approximately 200 samples/second. Because of `hpcrun`'s additional dynamic binary analysis, one might expect it to incur more overhead. However, our results show that a reasonable execution time and sampling rate quickly amortizes the binary analysis overhead over thousands of samples and makes it negligible.[4] In fact, the overhead differences between `hpcrun` and PTU are statistically insignificant. This is seen in two ways. First, the average overheads for each set of benchmarks are very similar; and given the high standard deviations, a statistical test would not meaningfully distinguish between the two. Second, average overheads for the individual benchmarks are within within 1-2% of each other, but no tool consistently performs better. Moreover, these small differences are well within the natural execution-time variability for a standard operating system (especially when using shared I/O) [23]; this fact accounts for the small *negative* overheads.

The one benchmark for which both `hpcrun` and PTU incur meaningful overhead is xalancbmk, at around 10%. The reason is that xalancbmk has many call paths that are 1000-2000 invocations long. An earlier version of `hpcrun` for the Alpha platform used a technique of inserting an 'active return' on a sample to memoize stack unwinds and collect return counts [9]. We plan to implement this technique and expect that it will significantly reduce `hpcrun`'s overhead in such cases.

2.3.2 Effectiveness

Given that `hpcrun` and PTU-smpl incur comparably low overheads, multi-column two of Table 1 assesses the quality of their call path profiles in terms of unwind failures. An unwind failure is defined as the inability to collect a complete calling context. Note that for `hpcrun`, this metric directly assesses the quality of unwind recipes and indirectly reflects the accuracy of procedure bounds. This is a reasonable metric because we have designed `hpcrun`'s binary analyses to cooperate for the purpose of obtaining accurate unwinds.

There are two ways to directly measure unwind failures. The most comprehensive method uses binary analysis to attempt to verify each link in the recovered call chain. For each each step in the unwind, we have a segment $p \rightarrow q$ and a return address (RA) within p. The analysis can then certify the unwind from q to p as (almost certainly) *valid*, *likely*, or (provably) *invalid*:

- *valid*, if a statically-linked call to q immediately precedes RA
- *valid*, if a dynamically-linked call to q immediately precedes RA (via inspection of the procedure linkage table)
- *likely*, if a dynamically-dispatched call immediately precedes RA
- *likely*, if a call to procedure r immediately precedes RA, and r is known to have tail calls
- *invalid*, if none of the above apply

Two details are worth noting. First, for architectures with variable-width instructions, it is reasonable to simply test offsets from RA that correspond to possible call or jump instructions rather than disassembling from the beginning of the procedure. Second, delay slots will offset the location of the call site.

The second way to measure unwind failures is based on the observation that, in practice, if an unwinder attempts to use an incorrect frame or stack pointer, errors very quickly accumulate and result in return addresses that are provably wrong in that they do not correspond to mapped code segments. Also, `hpcrun`'s program monitoring technology is able to intercept a process's or thread's entry point (for both statically and dynamically linked binaries). Thus, this second method classifies an unwind as invalid if it finds a provably wrong return address or if the unwind is not rooted in the process's or thread's entry point.

`hpcrun` currently implements the second method and discards all invalid unwinds. We are in the process of implementing the first, stronger version.

In contrast, for PTU-smpl, we measured unwind failures indirectly. PTU-smpl does retain partial unwinds; and if it performs any sort of verification, that information is not exported. Therefore, we wrote a script to analyze the results of PTU-smpl's 'hot path' listing. The script classifies a path as valid if it is rooted at some variant of "main" or any ancestor frame. Observe that this requirement is more relaxed than `hpcrun`'s. It is also worth noting that this requirement does *not* not penalize PTU-smpl for skipping a frame by incorrectly following its *parent*'s frame pointer rather than its own — an easy mistake for an x86-64 tool that is unwinding from an epilogue or frame-less procedure and that relies on compiler-generated unwind information.

Our results showed radically different failure rates for PTU-smpl on Intel-generated code (5%) versus PathScale and PGI code (65-75%). Since PTU-smpl is dependent upon frame pointers and unwind information, and since frame pointers are not reliably maintained in these binaries, the results strongly suggest that, compared to PathScale and PGI, the Intel compiler places a much higher priority on consistently recording correct unwind information. However, even on Intel-generated binaries, PTU-smpl can have high enough failure rates — as high as 5-20% — that it risks introducing systematic distortion by failing to unwind through a commonly appearing procedure instance. On the non-Intel benchmark versions, PTU-smpl's failure rate is so high that it essentially becomes a call path *fragment* profiler.

In contrast, the number of unwind failures for `hpcrun` is vanishingly small. `hpcrun`'s failures are reported as the average *number* (not percent) of failures over all six benchmark versions. Its worst performance was on the gcc benchmark. The benchmark averages on the order of 100K samples. Across the six versions of the benchmark that we studied, `hpcrun` failed to gather a full call path for 16 of those samples on average.

2.3.3 Summary

Despite the fact that `hpcrun`'s binary analysis for unwind recipes is a) context insensitive, b) operates without a control flow graph, c) does not formally track register values, and d) cannot treat embedded data as such, these results show that the cost of our

[4] Although it is more difficult to amortize the overhead of our binary analyses for very short executions, this does not imply that for such executions tools like PTU-smpl that use statically-computed unwind information induce significantly less overhead. Because typical compiler-generated unwind information is stored sparsely, a tool like PTU-smpl must invest some effort to read and interpret it.

```
(LM /mypath/hmc                                              load module
  (File /mypath/hmc.cc                                       source file
    (Proc doHMC 257-449 {[0xabe-0xfeed)}                     procedure
      (Stmt 309-309 {[bab1-0xbabe)} )                        statement
      (Loop 311-435 {[0xdad-0xfad)}                          loop
        (Stmt 313-313 {[0xdaf-0xea1), [ee1-0xeef)} )
      ))))
```

Figure 2. An object to source-code-structure map.

analysis is very modest and its results are very effective. Given that `hpcrun` almost always collects a full call path and that PTU-smpl much more frequently fails, we can say that on average `hpcrun` performs more useful work per sample than PTU-smpl — at the same overhead.

The clearest downside to our approach is the effort we have invested in developing these heuristics. The x86 unwinder was the most difficult to write, in large part because of its irregular architecture and variable-sized instructions. Nevertheless, once we arrived at the general approach we were able to relatively quickly develop MIPS and PowerPC unwinders. For example, we wrote the PowerPC unwinder — for use on Blue Gene/P — and resolved some OS-specific issues in about a week and a half. During our first major test, we collected performance data for an 8192-core execution of the FLASH astrophysics code [7] compiled with the IBM XL Fortran and C compilers for BG/P (versions 11.1 and 9.0, respectively) using options `-O4 -qinline -qnoipa`.[5] Out of approximately 1 billion total samples, `hpcrun` failed to unwind approximately 13,000 times — a failure rate of .0013%.

3. Binary Analysis for Source-Level Attribution

To combine dynamic call path profiles with the static structure of fully optimized binaries, we need a mapping between object code and its associated source code structure. An example of what this mapping might look like is shown in Figure 2. The mapping is represented as a scope tree, where a load module (a binary) contains source files; files contain procedures; procedures contain loops; procedures and loops contain statements; and scopes such as procedures, loops and statements can be annotated with object code address interval sets.

There are two ways to obtain the desired mapping: use a summary of transformations recorded by the compiler or reconstruct it through analysis. Because debuggers must associate the execution of object code with source code, one would expect debugging information to provide the former. In 1992, Brooks *et al.* [3] developed debugging extensions for mapping object code to a scope tree of procedures, loops, blocks, statements and expressions. While they left to future work a solution for the inlining problem, neither compilers nor debugging formats followed their lead. Although DWARF [8], the *de facto* standard on Linux, can represent inlining, it cannot describe loops or loop transformations. Even worse, all x86 Linux compilers that we have used generate only limited DWARF, often failing to record inlining decisions. Intel's compiler (10.x) retains line-level information in the presence of inlining, but the information is incomplete (*e.g.*, there is no association between inlined code and object code) and sometimes erroneous. Thus, however easy the problem of creating the object to source code mapping could have been, the fact remains that vendor compilers do not provide what we desire. Consequently, we wrote the `hpcstruct` tool to reconstruct the mapping through binary analysis, using only a 'lowest common denominator' set of debugging information. We focus on programs written in C++, C, and Fortran.

[5] We were forced to disable inter-procedural analysis because of an incompatibility between IBM's compiler and our tool for inserting `hpcrun` in statically-linked binaries.

Address	File	Line	Procedure
0x...15550	hmc.cc	499	main
0x...15570	hmc.cc	14	main
0x...17030	qdp_multi.h	35	main
0x...172c0	stl_tree.h	1110	main

Figure 3. Typical line map information.

An obvious starting point is to consult an executable's line map, which maps an object address to its corresponding source file, line number and procedure name for use by a debugger. However, the line map is insufficient for detecting inlined, or more generally, *alien* code, *i.e.*, code that originates outside of a given procedure. To see this, consider the unexceptional line map excerpt from a quantum chromodynamics code shown in Figure 3. Given that the first entry maps to native (as opposed to alien) code, what is the first line of procedure `main`? Although one is tempted to answer 14, it turns out that the second line is actually alien; this is not detectable because the line map retains the *original* file and line information (from *before* inlining) but assumes the name of the host procedure (*after* inlining). Even worse, because optimizing compilers reorder the native and alien instructions (including prologues and epilogues), no particular entry is guaranteed to map to native code, much less the procedure's begin or end line. Consequently, to reconstruct the desired mapping we must supplement the line map with a 'lowest common denominator' set of DWARF-specific information.

3.1 Recovering the Procedure Hierarchy

Compilers perform several procedure transformations such as flattening nested procedures, inlining, and cloning for specialization. Recovering the procedure hierarchy involves re-nesting source code procedure representations, determining their source line bounds and identifying alien code.

It turns out that by combining standard DWARF information with certain procedure invariants, recovering the procedure hierarchy is less difficult than it first appears. A load module's DWARF contains procedure descriptors for each object procedure in the load module *and* the nesting relationship between the descriptors. Each descriptor includes 1) the procedure's name, 2) the defining source file and begin line, and 3) its object address ranges. The key missing piece of information is the procedure's end line. Observe however, that two source procedures do not have overlapping source lines unless they are the same procedure or one is nested inside the other. Intuitively, in block structured languages, source code does not 'overlap.' More formally:

Non-overlapping Principle. *Let scopes x_1 and x_2 have source line intervals σ_1 and σ_2 within the same file. Then, either x_1 and x_2 are the same, disjoint or nested, but not overlapping:*[6]

- $(x_1 = x_2) \Leftrightarrow (\sigma_1 = \sigma_2)$
- $(x_1 \neq x_2) \Leftrightarrow ((\sigma_1 \cap \sigma_2 = \emptyset) \vee (\sigma_1 \subset \sigma_2) \vee (\sigma_2 \subset \sigma_1))$

We can also say (where $x_2 \sqsubset\!\!\!< x_1$ means x_1 is nested in x_2):

- $(\sigma_1 \cap \sigma_2 = \emptyset) \Leftrightarrow ((x_1 \neq x_2) \wedge \neg(x_1 \sqsubset\!\!\!< x_2) \wedge \neg(x_2 \sqsubset\!\!\!< x_1))$
- $(\sigma_2 \subset \sigma_1) \Leftrightarrow (x_1 \sqsubset\!\!\!< x_2)$

The implication of this principle is that given DWARF nesting information, we can infer end line bounds for procedures, resulting in the following invariants:

Procedure Invariant 1. *A procedure's bounds are constrained by any (parent) procedures that contain it.*

[6] Unstructured programming constructs may give rise to irreducible loops or alternate procedure entries. While the former is not strictly an exception (no block of source code actually overlaps), the latter is. However, Fortran's alternate entry statement is deprecated and used very infrequently.

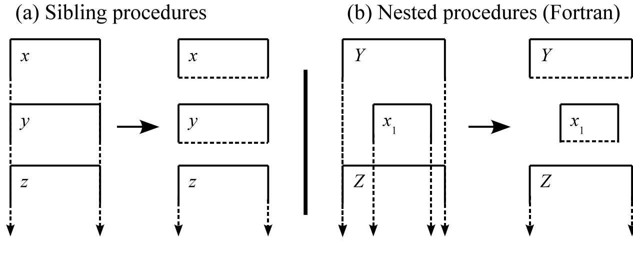


Figure 4. Bounding procedure end lines.

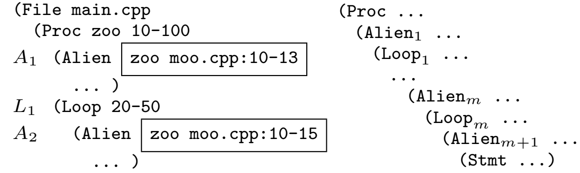


Figure 5. (a) Alien context ambiguity; (b) Maximum procedure context nesting for scope s.

Procedure Invariant 2. *Let procedure y have sibling procedures x and z before and after it, respectively. Then, y's begin line is greater than x's end line and its end line is less than z's begin line.*[7] Figure 4a graphically depicts application of this invariant.

Neither C++ nor C permits procedure nesting. To handle Fortran, which places strict limits on where a procedure can be nested, we derive a special invariant (depicted graphically in Figure 4b):[8]

Procedure Invariant 3. *Let procedure Y have nested procedures $x_1 \dots x_n$, in that order. Then Fortran nesting implies that the executable code of Y and $x_1 \dots x_n$ forms $n+1$ ordered, contiguous source code regions.*

These invariants enable `hpcstruct` to infer an upper bound on all procedure end lines except for the last top-level procedure of a source file, whose upper bound is ∞. Moreover, accurate procedure bounds information is sufficient for detecting *all* alien code within a procedure (assuming two restrictions discussed below).

There are two complications with this strategy. First, it is often the case that a load module's DWARF does not contain a DWARF descriptor for every source level procedure, creating 'gaps' in the procedure hierarchy. For example, no descriptor is generated for a C++ static procedure inlined at every call site. Although this knowledge can never be fully recovered, we have developed a simple and effective heuristic to close most of the important gaps [28].

Second, C++ permits classes to be declared within the scope of a procedure, thereby allowing class member functions to be transitively nested within that procedure. Consider a procedure-scoped C++ class with n member functions. The nth member function may be inlined into the procedure but because the only end line bound we can establish on the nth member function is the end line bound of the containing procedure itself, we will not be able to detect it. This means that in the presence of procedure-scoped classes, *even* with DWARF descriptors for every procedure we may not be able to detect all alien code. However, this issue is of little practical concern: procedure-scoped classes are rare; and we have developed a strategy for detecting the presence of most procedure-scoped classes [28].

A high-level sketch of `hpcstruct` is shown in Algorithm 2. It consists of two parts: recovering the procedure hierarchy and recovering loop nests for each procedure. This section has covered the first part; the second part is covered below.

3.2 Recovering Alien Contexts

Before discussing loops, we note three important aspects of detecting alien code.

Figure 5a shows an example of two alien scopes, A_1 and A_2, representing the presence of alien code within procedure `zoo`. Consider the task of identifying the alien code within `zoo`. In general, given an object code instruction, its corresponding source level statement is classified as alien if its source file is different than the enclosing procedure's or if its source line is outside the line bounds of the enclosing procedure's. However, as an instruction is processed, adjacent instructions may belong to different alien contexts (*i.e.*, different inlined procedures). Since inlining can be nested, it is natural to ask how to distinguish between nested and non-nested inlining. The short answer is that without DWARF inlining or source-level call graph information, we cannot. Therefore, we choose to flatten alien scopes with respect to their enclosing loop or procedure. This implies that for a loop nest of depth m, there can be at most $m+2$ parent contexts (procedure or alien scopes), as illustrated in Figure 5b.

Return again to Figure 5a. Observe that A_1 and A_2 have overlapping bounds, where A_2 is embedded within loop L_1. Without call site information, it is not possible to distinguish between 1) one distinct call site within the loop, where some of the inlined code was was loop invariant; or 2) two distinct call sites where some of the code from the first call site (A_1) was entirely eliminated.

Finally, the number and bounds of alien scopes can be refined using the Non-overlapping Principle [28].

3.3 Recovering Loop Nests

Having an outline of the procedure hierarchy, `hpcstruct` recovers the loop nesting structure for each procedure. As shown in Algorithm 2, this task can be broadly divided into two components: 1) analyzing object code to find loops (line 6) and 2) inferring a source code representation from them (line 7). To find loop nests within the object code, `hpcstruct` first decodes the machine instructions in a procedure to compute the control flow graph (CFG) and then uses Havlak's algorithm [13] to recover the tree of loop nests [19]. Given this tree of object code loops, `hpcstruct` then recovers a source code representation for them. This is a challenging problem because with fundamentally line-based information `hpcstruct` must distinguish between 1) loops that contain inlined code, 2) loops that may themselves be inlined, and 3) loops that may be inlined *and* contain inlined code. Finally, `hpcstruct` must account for loop transformations such as software pipelining.

Because loops also obey the Non-overlapping Principle, there are analogous loop invariants for Procedure Invariants 1 and 2. However, without symbolic loop information, these invariants are of little value. Consequently, `hpcstruct`'s strategy is to initially assume that the source loop nesting tree mirrors the object code loop tree, and then look for exceptions. Specifically, `hpcstruct` performs a preorder traversal of the object loop tree, recursively visiting outer loops before inner loops. The challenge we now discuss is reconstructing a source representation for every loop during this traversal.

As a starting point, we observe that loop invariant code motion implies that *a computation at loop level l will (usually) not be moved into a loop that is at a nesting level deeper than l.* Coupling this observation with accurate procedure bounds, we could scan through all the non-alien statements within a particular loop and compute a minimum and maximum line number, which we call the min-max heuristic.

[7] We can ignore the case where two procedures are defined on the same source line; column information would make this precise.

[8] Because DWARF contains a language identifier, this nesting rule can be applied only when appropriate.

Algorithm 2: High-level sketch of recovering a binary's static source code structure.

Input: A load module *lm* (with DWARF information)
Result: $\mathcal{S}$, *lm*'s object to source code structure map

let $\mathcal{D}$, dwarf map : *object-procedure* $\mapsto$ *DWARF-descriptor*
let $\mathcal{L}$, line map : *address* $\mapsto$ $\langle$*file-name, proc-name, line*$\rangle$

`// Recover procedure hierarchy (§3.1)`
Create a source procedure $p_{\mathcal{S}}$ for each DWARF descriptor in $\mathcal{D}$ with no object code
Create a source procedure $p_{\mathcal{S}}$ for each object-procedure $p_{\mathcal{O}}$ using $\mathcal{D}(p_{\mathcal{O}})$ or $\mathcal{L}(p_{\mathcal{O}})$.

`// Recover loop nests (§3.3)`
foreach *procedure* $p_{\mathcal{S}}$ *in* $\mathcal{S}$ *with object-procedure* $p_{\mathcal{O}}$ **do**
6 Form $p_{\mathcal{O}}$'s loop nests by creating the strongly connected regions tree T induced by $p_{\mathcal{O}}$'s control flow graph
7 **foreach** *basic block* b *in* T *(preorder traversal)* **do**
  if b *is a loop header* **then**
   let $\sigma = \mathcal{L}(i)$ for backward-branch i
   let $es_{\mathcal{S}} = \mathsf{determine\text{-}context}(\sigma)$
   Create a source code loop $l_{\mathcal{S}}$ located within $es_{\mathcal{S}}$
  foreach *instruction* i *in* b **do**
   let $\sigma = \mathcal{L}(i)$
   let $es_{\mathcal{S}} = \mathsf{determine\text{-}context}(\sigma)$
   Create a statement scope $s_{\mathcal{S}}$ for σ within $es_{\mathcal{S}}$
Normalize each procedure p in $\mathcal{S}$ (§3.4)

```
                                                      Steps
   (File main.cpp
     (Proc init 145-199
A1   (Alien ... Array.cpp:82-83                       1. Find alien context
S1     (Stmt 82-82)
L2     (Loop 83-83                                    2. Locate loop (incorrectly)
S2       (Stmt 83-83)
A3       (Alien ... [main.cpp]:158-158
S3         [(Stmt 158-158)]                           3. [Self nesting!]
```

Figure 6. Detecting incorrect loop placement via nesting cycles.

One complication for the min-max heuristic is Fortran's use of *statement functions*, which are single-statement functions nested within a procedure. Statement functions have no associated DWARF descriptors. Code for statement functions is forward substituted wherever they are used. Applying the min-max heuristic to the first loop of a procedure that uses a statement function will result in a loop begin line that erroneously includes all executable statements prior to the loop. To prevent this problem, we would like some mechanism for estimating the begin line of a loop. When loops are compiled to object code, the loop header's continuation test is typically translated into a conditional backward branch that, based on the result of the continuation test, returns to the top of the loop or falls through to the next instruction. Moreover, most compilers associate the loop's backward branch with the source line of the continuation test, and therefore the loop header. We therefore modify the simple min-max heuristic to form the bbranch-max heuristic for computing loop begin and end lines: the loop begin line can be approximated using information from the backward branch; and the best loop end line is the maximum line after all alien lines have been removed.

Although the bbranch-max heuristic can be thwarted by unstructured control flow, it suffers from a more serious defect. The difficulty is that when estimating a loop's begin line from that loop's continuation test, the heuristic implicitly determines the loop's *procedure context*, *i.e.*, the loop's enclosing alien or procedure scope. Specifically, bbranch-max assumes that the procedure context for that instruction is the same context as other instructions within the (object) loop body. This results in a severe problem if the loop's condition test derives from inlined code, something that is *very* common within object-oriented C++. Therefore, it is necessary to somehow distinguish between a loop deriving from an alien context (and which itself may have alien loops) and one that only contains alien contexts within its header or body. As previously suggested, our solution to this problem, is to *guess and correct*. In brief, `hpcstruct` processes instructions within a loop one-by-one (Algorithm 2, line 7); and for each instruction it determines that instruction's procedure context, its source line location within that context, and its enclosing loop (if any). Figure 6 shows a partially reconstructed procedure where alien scope A_1 has been identified (Step 1) by using the source line information for the instruction corresponding to S_1. When `hpcstruct` processes the loop header (S_2) for L_2 using bbranch-max (Step 2), it must determine whether the source line loop should be located in the current procedure context, a prior context (which would imply the current context is alien), or a new alien context. In this case, because of the presence of statement S_2, `hpcstruct` 'guesses' that the loop header should be located within the current alien procedure context A_1. `hpcstruct` next processes S_3 (Step 3), which it determines must be alien to the current procedure context A_1, resulting in the new alien context A_3. However, because A_3's bounds are within `init`'s bounds, this implies that `init` is inlined inside of itself, which is a contradiction. This shows that the guess at Step 2 was wrong.

```
    Before                                  After
    (File main.cpp                          (File main.cpp
      (Proc init 145-199                      (Proc init 145-199
A1    (Alien Array.cpp:82-83>                   (Alien Array.cpp:82-83
        (Stmt 82-82)                              (Stmt 82-82)
                                                )
L1      (Loop 83-83)                            (Loop [158-158]
                                                  (Alien Array.cpp:82-83
S2        (Stmt 83-83)                              (Stmt 83-83)
                                                  )
          (Alien main.cpp:158-158
S3          (Stmt [158-158])                      (Stmt 158-158)
```

Figure 7. Correcting nesting cycles.

This observation, which is another implication of the Non-overlapping Principle, can be formally stated as follows:

Procedure Invariant 4. *Let L be a loop nest rooted in an alien scope C_a. Furthermore, let L have loop levels $1 \ldots n$. Now, let s be a statement at level n that clearly belongs in a shallower procedure context C'. Since C' is a shallower procedure context, it must be a parent of C_a which implies that C' is nested within itself, which is impossible.*

When an impossibility such as this is found, `hpcstruct`, knowing that L was mislocated, corrects the situation by relocating all levels of L from C_a to within C'. Figure 7 shows how we correct the loop nesting cycle shown in Figure 6. In this case, L_1 is un-nested one level, which places it within the correct procedure context and its bounds are updated to include S_3. S_2 remains nested in L_1, but A_1's context must be replicated to correctly represent it. (The fact that a loop nest of depth m can have at most $m + 2$ parent contexts bounds the cost of this correction process in practice.)

Observe that to properly recover the corrected L_1, it is critical to appropriately expand its begin line so that statements that should belong in the loop are not ejected. To do this, we use a tolerance factor when testing for a statement's inclusion within the current loop. If the current begin line minus the tolerance factor would include the statement within the bounds, the statement is deemed to

be within the loop and the bounds grow accordingly; the loop's end line can thought of having a tolerance of ∞ to assign the maximum line within the loop as the end line. The effects of fuzzy matching can be complex, because a loop may initially appear to be within an alien context (by backward branch information) but later emerge as a native loop. To account for this, `hpcstruct` uses different tolerances based on context [28].

3.4 Normalization

Because of loop transformations such as invariant code motion and software pipelining, the same line instance may be found both within and outside of a loop or there may be duplicate nests that appear to be siblings. To account for such transformations, we developed normalization passes based on the observation that *a particular source line (statement) appears uniquely with a source file* (an application of the Non-overlapping Principle). For its most important normalization passes, `hpcstruct` repeatedly applies the following rules until a fixed point is reached:

- Whenever a statement instance (line) appears in two or more disjoint loop nests, fuse the nests *but only within the same procedure context*. (Correct for loop splitting.)
- Whenever a statement instance (line) appears at multiple distinct levels of the same loop nest (i.e., *not crossing procedure contexts*), elide all instances other than the most deeply nested one. (Correct for loop-invariant code motion.)

3.5 Summary

Thorough application of a small set of invariants enables `hpcstruct` to recover very accurate program structure even in the presence of complex inlining and loop transformations. Importantly, in the (rare) worst case, while the effects of an incorrect inference may be compounded, they are limited to at most *one* procedure. Further details, including discussions of macros, procedure groups and algorithms can be found in [28].

We have tested `hpcstruct` on the GCC, Intel, PathScale, Portland Group and IBM XL compilers (among others). When debugging information is accurate, `hpcstruct` produces very good results. However, we have observed that debugging information from certain compilers is sometimes erroneous — and even violates the DWARF standard. We have hardened `hpcstruct` to handle certain errors, but it cannot psychoanalyze. While compilers may opt to generate incomplete information, the information that they do generate should be *correct*.

4. Putting It All Together

By combining `hpcrun`'s minimally intrusive call path profiles and `hpcstruct`'s program structure, we relate execution costs for a fully optimized executable back to static and dynamic contexts overlaid on its source code. To demonstrate our tools' capabilities for analyzing the performance of modular applications, we present screen shots of HPCTOOLKIT's `hpcviewer` browser displaying performance data collected for two modern scientific codes.

4.1 MOAB

We first show the detailed attribution of performance data for MOAB, a C++ library for efficiently representing and evaluating mesh data [29]. MOAB implements the ITAPS iMesh interface [16], a uniform interface to scientific mesh data. We compiled MOAB on an AMD Opteron (Barcelona) based system using the Intel 10.1 compiler with `-O3`. (We could not use `-fast` because of a compiler error.) We profiled a serial execution the `mbperf` performance test using a $200 \times 200 \times 200$ brick mesh and the array-based/bulk interface.

Figure 8(a) shows a calling context tree view of a call path profile of MOAB. The navigation pane (lower left sub-pane) shows a partial expansion of the calling context tree. The information presented in this pane is a fusion of `hpcrun`'s dynamic and `hpcstruct`'s static context information. The selected line in the navigation pane (at the bottom) corresponds to the highlight in the source pane (top sub-pane).

The navigation pane focuses on the hottest call path (automatically expanded by `hpcviewer` with respect to L1 data cache misses). A closer look reveals that the path contains *six* loops dynamically nested within inlined and non-inlined procedure activations. The root of the path begins prosaically with `main` $\rightarrow$ `testB` but then encounters an inlined procedure and loop from `mbperf_iMesh.cpp`. The inlined loop makes a (non-inlined) call to `imesh_getentadj` which descends through several layers of mesh iteration abstractions. Near the end of the hot call path, `AEntityFactory::get_adjacencies` contains an inlined code fragment from the C++ Standard Template Library (STL), which itself contains a loop over code inlined from the MOAB application (TypeSequenceManager.hpp). Closer inspection of the call path confirms that `get_adjacencies` calls an (inlined) procedure that calls the STL `set::find` function — which makes a call back to a user-supplied comparison functor in TypeSequenceManager.hpp. In this context, the comparison functor incurs 21.3% of all L1 data cache misses, suggesting that objects in the STL set should be allocated to exploit locality. Our tools are uniquely able to attribute performance data at the source level with exquisite detail, even in the presence inlining.

4.2 S3D

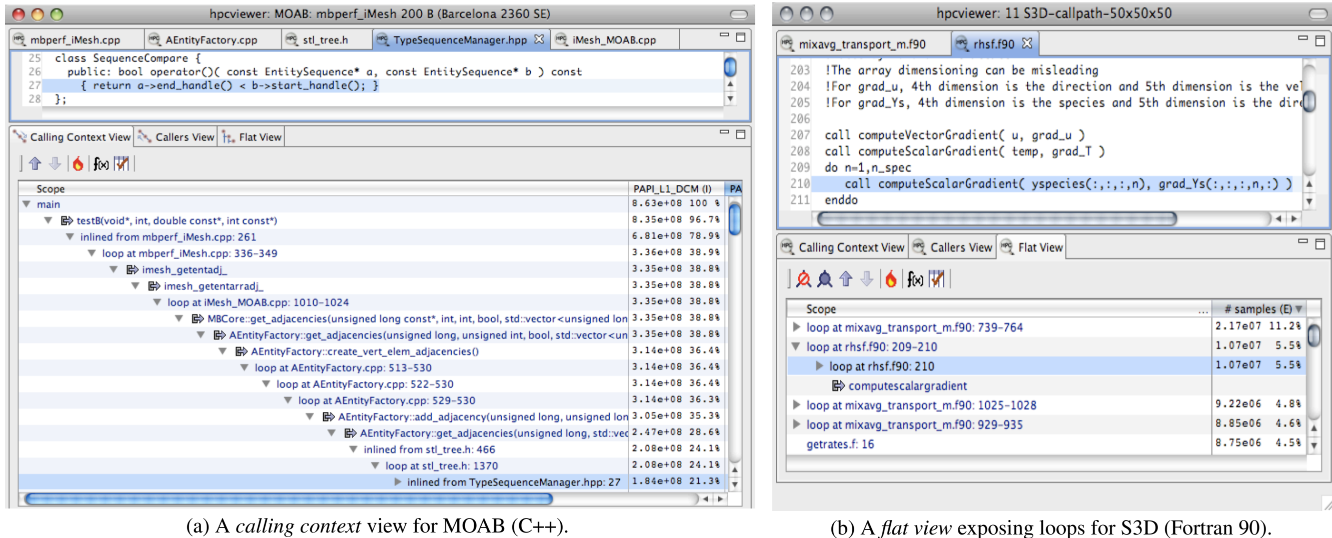
The second application we discuss is S3D, a Fortran 90 code for high fidelity simulation of turbulent reacting flows [20]. We compiled S3D on a Cray XD1 (AMD Opteron 275) using Portland Group's 6.1.2 compiler with the `-fast` option.

Figure 8(b) shows part of a loop-level 'flat view' for a call path profile of a single-core execution. The flat view organizes performance data according to an application's static structure. All costs incurred in any calling context by a procedure are aggregated together in the flat view. This particular view was obtained by flattening away the procedures normally shown at the outermost level of the flat view to show outer-level loops. This enables us to view the performance of all loop nests in the application as peers. We focus on the second loop on lines 209-210 of file `rhsf.90`. Notice that this loop contains a loop at line 210 that does not appear explicitly in the code. This loop consumes 5.5% of the total execution time. This is a compiler-generated loop for copying a non-contiguous 4-dimensional slice of array `grad_Ys` into a contiguous array temporary before passing it to `computeScalarGradient`. The ability to explicitly discover and attribute costs to such compiler-generated loops is a unique strength of our tools.

5. Considering Other Contemporary Tools

There is a large body of prior work on call path profiling, but its focus has not been on using binary analysis to enable sampling-based measurement and attribution of performance metrics for fully optimized code. For this this reason we focus on comparing with contemporary tools with the most closely related capabilities for measurement and attribution.

To our knowledge, no other sampling based profiler is capable of collecting full call path profiles for fully optimized code. Any tool based on libunwind [21] such as LoopSampler [22] requires frame pointers or unwind information. OProfile [17] and Sysprof [25], two well-known Linux system-wide call stack profilers require frame pointers. Since the x86-64 ABI does not require frame pointers, this restriction requires recompilation of any appli-



(a) A *calling context* view for MOAB (C++).

(b) A *flat view* exposing loops for S3D (Fortran 90).

Figure 8. `hpcviewer` presenting different views of call path profiles for two applications.

cation and system library of interest. Apple's Shark [2], one of the nicer tools, also fails to correctly unwind optimized code. On a simple test, we observed it incorrectly unwinding calls from the `sinh` math library procedure.

Sampling-based call path profilers naturally fail to record a complete calling context tree. However, they also naturally highlight the 'most important' paths, which comports well with performance analysis. Zhuang *et al.* develop 'bursty' call path profiling for Java [31] — a combination of sampling and adaptive, time-limited dynamic instrumentation — that more accurately approximates the complete CCT with an average overhead of 20%. For performance tuning, it is no bargain to pay such overhead to increase the knowledge of infrequently executed paths.

The importance of correlating performance measurements with source code has been widely acknowledged. The task of correlation is easy with custom-generated compiler information [1, 30]. Unfortunately, this solution is impractical. Typically, open systems supply multiple compilers. Consequently, current sampling-based call path profilers trivially correlate dynamic data with source code using the binary's line map. In the presence of inlining and loop transformations, this approach results in confusing correlations that attribute costs of inlined code back to their source files rather than where they were incurred.

6. Conclusions

We have designed methods of binary analysis for 1) minimally intrusive call path profiling of fully optimized code and 2) effective attribution and interpretation of performance measurements of fully optimized code. Our evaluation of `hpcrun` using the SPEC benchmarks on executables optimized by several different compilers shows that we can attribute costs incurred by fully optimized code to full calling context with low run-time overhead. The examples in Figure 8 highlight the unique contextual information we obtain by combining `hpcrun`'s dynamic call path information with `hpcstruct`'s static program structure. They show both how we attribute costs to inlined frames and loop nests and how this information yields insight into the performance of complex codes.

When compared with instrumentation-based techniques, our measurement and analysis methods have several advantages. *First*, sampling-based call path profilers do not interfere with compiler optimization and introduce minimal distortion during profiling. On many operating systems, they can even be invoked on unmodified dynamically linked binaries. *Second*, using binary analysis to recover source code structure is uniquely complementary to sampling-based profiling. `hpcrun` samples the whole calling context in the presence of optimized libraries and even threads. `hpcstruct` recovers the source code structure, by using only minimal symbolic information, for any portion of the calling context — even without the source code itself. Using binary analysis to recover source code structure addresses the complexity of real systems in which source code for libraries is often missing. *Third*, binary analysis is an effective means of recovering the source code structure of fully optimized binaries. When source code is available, we have seen that `hpcstruct`'s object to source code structure mapping accurately correlates highly optimized binaries with procedures and loops. Among other things, it accounts for inlined routines, inlined loops, fused loops, and compiler generated loops. In effect, our binary analysis methods have enabled us to observe both what the compiler did and did *not* do to improve performance. We conclude that our binary analyses enable a unique combination of call path data and static source code structure; and this combination provides unique insight into the performance of modular applications that have been subjected to complex compiler transformations.

Both of our analyses have been motivated, in part, by a lack of compiler information. While we would welcome improved compiler support, it seems unlikely any will be forthcoming. Although compiler vendors have been sympathetic to our requests to fix or improve their symbolic information, they have been clear that their highest priority is highly efficient and correct code. Improving line maps or debugging information in binaries is at the bottom of their list of tasks. We have shown that accurate and rich contextual information can be obtained with only minimal compiler information and we believe that the utility of our results justify our effort.

Acknowledgments

Development of HPCToolkit is supported by the Department of Energy's Office of Science under cooperative agreements DE-FC02-07ER25800 and DE-FC02-06ER25762. HPCToolkit would not exist without the contributions of the other project members: Laksono Adhianto, Mark Krentel, and Gabriel Marin. Mark Krentel's efforts have vastly improved `hpcrun`'s ability to dynamically and statically monitor processes and threads. HPCToolkit's `hpcviewer` interface is primarily the work of Laksono Adhianto.

We are grateful to Mark Charney and Robert Cohn at Intel for assistance with XED [5]. This work used equipment purchased in part with funds from NSF Grant CNS-0421109.

References

[1] V. S. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. Kennedy. An integrated compilation and performance analysis environment for data parallel programs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 50, New York, NY, USA, 1995. ACM Press.

[2] Apple Computer. Shark. `http://developer.apple.com/tools/sharkoptimize.html`.

[3] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 1992. ACM Press.

[4] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] M. Charney. XED2 user guide. `http://www.pintool.org/docs/24110/Xed/html`.

[6] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 80–89, Washington, DC, USA, 1996. IEEE Computer Society.

[7] A. Dubey, L. Reid, and R. Fisher. Introduction to FLASH 3.0, with application to supersonic turbulence. *Physica Scripta*, 132:014046, 2008.

[8] Free Standards Group. DWARF debugging information format, version 3. `http://dwarf.freestandards.org`. 20 December, 2005.

[9] 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.

[10] N. Froyd, N. Tallent, J. Mellor-Crummey, and R. Fowler. Call path profiling for unmodified, optimized binaries. In *GCC Summit '06: Proceedings of the GCC Developers' Summit, 2006*, pages 21–36, 2006.

[11] 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.

[12] R. J. Hall. Call path profiling. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 296–306, New York, NY, USA, 1992. ACM Press.

[13] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[14] Intel Corporation. Intel performance tuning utility. `http://software.intel.com/en-us/articles/intel-performance-tuning-utility`.

[15] Intel Corporation. Intel VTune performance analyzer. `http://www.intel.com/software/products/vtune`.

[16] ITAPS working group. The ITAPS iMesh interface. `http://www.tstt-scidac.org/software/documentation/iMesh_userguide.pdf`.

[17] J. Levon *et al.* OProfile. `http://oprofile.sourceforge.net`.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[19] 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(1):81–104, 2002.

[20] D. Monroe. ENERGY Science with DIGITAL Combustors. `http://www.scidacreview.org/0602/html/combustion.html`.

[21] D. Mosberger-Tang. libunwind. `http://www.nongnu.org/libunwind`.

[22] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 143–152, New York, NY, USA, 2007. ACM.

[23] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. Producing wrong data without doing anything obviously wrong! In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, 2009.

[24] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)*, pages 798–804, 2008.

[25] S. Sandmann. Sysprof. `http://www.daimi.au.dk/~sandmann/sysprof`. 21 October 2007.

[26] S. S. Shende and A. D. Malony. The Tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, 2006.

[27] SPEC Corporation. SPEC CPU2006 benchmark suite. `http://www.spec.org/cpu2006`. 3 November 2007.

[28] N. R. Tallent. Binary analysis for attribution and interpretation of performance measurements on fully-optimized code. M.S. thesis, Department of Computer Science, Rice University, May 2007.

[29] T. J. Tautges. MOAB-SD: integrated structured and unstructured mesh representation. *Eng. Comput. (Lond.)*, 20(3):286–293, 2004.

[30] 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.

[31] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 263–271, New York, NY, USA, 2006. ACM.