Low Overhead Program Monitoring and Profiling

Naveen Kumar[†], Bruce R. Childers[†], and Mary Lou Soffa[‡]

[†]Department of Computer Science
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
{naveen, childers}@cs.pitt.edu

[‡]Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
soffa@virginia.edu

Abstract

Program instrumentation, inserted either before or during execution, is rapidly becoming a necessary component of many systems. Instrumentation is commonly used to collect information for many diverse analysis applications, such as detecting program invariants, dynamic slicing and alias analysis, software security checking, and computer architecture modeling. Because instrumentation typically has a high run-time overhead, techniques are needed to mitigate the overheads. This paper describes "instrumentation optimizations" that reduce the overhead of profiling for program analysis. Our approach applies transformations to the instrumentation code that reduce the (1) number of instrumentation points executed, (2) cost of instrumentation probes, and (3) cost of instrumentation payload, while maintaining the semantics of the original instrumentation. We present the transformations and apply them for program profiling and computer architecture modeling. We evaluate the optimizations and show that the optimizations improve profiling performance by 1.26–2.63x and architecture modeling performance by 2–3.3x.

Categories and Subject Descriptors

D.2.5. [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.3.3. [**Programming Languages**]: Language Constructs and Features—*Program instrumentation*, *run-time environments*

General Terms

Languages, Performance, Algorithms

Keywords

Dynamic Binary Translation, Dynamic Instrumentation, Instrumentation Optimization, Profiling

1. Introduction

Instrumentation, used to profile and monitor a program, has received much attention due to the increased usefulness of information about a program's execution to analysis applications. Instrumentation is used in many settings for both static and dynamic analysis. Instrumentation is used by Daikon to discover program invariants [9] and by the dynamic code optimizers Jikes [1] and Dynamo [2] to guide code transformations. Other uses include dynamic program slicing [28], demand-driven structural software testing [18], modeling computer architecture features [4,16,23,26], and software security [12,22].

Many techniques have been proposed for instrumenting code. These techniques include placing instrumentation in a program before it executes, static binary rewriting for profiling [5,15,19,20,24] and placing (and removing) instrumentation in executing programs [1,10,17,19]. There are also infrastructures that provide instrumentation capabilities for different machine platforms [5,10,15,19]. An important aspect is how these approaches address instrumentation overhead. Tools such as ATOM apply context-specific optimizations to the instrumentation to reduce its cost [24]. However, ATOM does not reduce the number of locations where instrumentation is inserted, but can only reduce the cost of an individual instrumentation location. ATOM is also a static tool: It relies on expensive interprocedural analysis which is done at link-time, and thus, it is not suited for dynamic instrumentation. Other frameworks such as DynInst [10] and Pin [19] have efficient mechanisms for dynamic instrumentation, but they do not automatically apply instrumentation optimizations to mitigate overhead. Arnold and Ryder described a technique to conditionally execute instrumented code for dynamic instrumentation [1]. Although their approach uses a conditional probe to reduce overhead, it does not use optimizations at the instrumentation algorithm level. In these approaches it is left to the instrumentation application to determine how best to instrument a program, given low cost instrumentation probes and mechanisms.

Yet, both an opportunity and need exist to automatically apply optimizations on the instrumentation code to reduce its overhead, similar to code optimizations but targeted and specialized to instrumentation code. Our research is developing instrumentation techniques and optimizations that are targeted to reducing the overhead of dynamically instrumented code. This paper describes instrumentation optimizations and an instrumentation optimizer for program monitoring and profiling. Our initial instrumentation optimizations target three sources of overhead, including the (1) cost of instrumentation probes that intercept program execution, (2) the number of probes executed ("dynamic probe count"), and (3) the cost of instrumentation code itself (the "instrumentation payload"). One optimization minimizes the number of instrumentation points executed using coalescing while another optimization uses partial inlining to select

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PASTE'05, September 5–6, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-239-9/05/0009...$5.00.

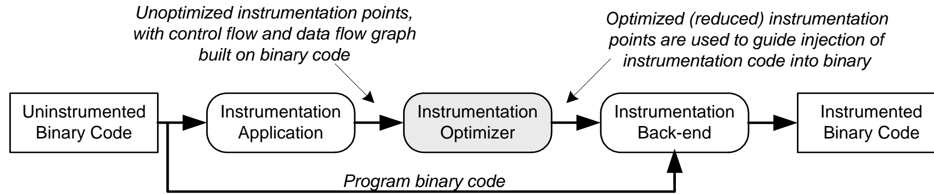

Figure 1: Approach to instrumentation optimization

between lightweight and heavyweight versions of instrumentation code as run-time conditions warrant. Finally, a third optimization uses specialization, including partial register context save and restore to make the instrumentation code more efficient. These optimizations can use dynamic information, such as program paths, to improve performance.

This paper first describes the instrumentation optimizations, and then demonstrates their usefulness in two case studies. The first case study uses the optimizations to improve the performance of lightweight program profilers. Profiles are typically used to gather information needed for program analysis. The second case study uses instrumentation optimization to improve the performance of online modeling of computer architecture features. Such modeling is used in the computer architecture community to analyze the performance of new architectural features.

This paper makes several contributions, including:

- Instrumentation optimizations that mitigate instrumentation overhead by reducing the number of probes, the cost of each probe and the cost of the payload code;
- An instrumentation optimizer, called INS-OP, that applies instrumentation optimizations in a dynamic binary code translator;
- A case study that demonstrates that the optimizations can improve the performance of lightweight profilers for basic block counts, dynamic call chains, program paths, and data and address values; and
- A second case study that applies INS-OP to computer architecture modeling and evaluates the optimizations on a fast cache simulator.

2. Instrumentation and Overhead

Because instrumentation is not normally part of a program, it incurs an overhead that is the extra amount of time spent in executing instrumentation code, rather than the application code. Instrumentation is typically inserted at specific points in the program, which we call *instrumentation points (IP)*. An IP encapsulates the instrumentation functionality and its program context. An IP consists of an *instrumentation probe* and *instrumentation payload*. The probe is the code that is inserted in the application to intercept program execution to invoke the payload. The payload does the actual instrumentation activity, such as collecting a profile. Because it may be unnecessary to invoke the payload on every execution of a probe, the probe can have a condition that controls when the payload is invoked. For example, such conditions can be used to implement sampling—when a count is within a specified range, profiling is done.

Instrumentation overhead is affected by the number of probes executed, or *the probe count*. The probe count for an IP determines the number of instances of the probe (i.e., the number of times that probe is hit). Every time a probe is hit, it incurs an overhead associated with executing the probe code that intercepts program execution. Because the payload can be guarded by a condition, every instance of a probe may not incur the overhead of the payload. Thus, the total overhead for an IP is related to the number of probes executed, how much each probe costs, how frequently the payload is invoked and the cost of the payload.

Instrumentation optimizations can be targeted to each overhead component. To reduce overhead, the optimizations transform the instrumentation code into a more efficient form. Transformations are applied to a set of unoptimized IPs to get optimized IPs. The transformations reduce the number of run-time instances by combining and eliminating some IPs, which leads to fewer overall executed instances of IPs. Transformations are done on the instrumentation code itself to reduce probe and payload cost. Partial inlining can be applied to reduce the payload cost. In this case, a lightweight version of the instrumentation payload may be incorporated directly into the application binary to gather minimal information and perform a small set of actions. When necessary, the full version (i.e., a heavyweight version) of the payload is invoked. For example, the number of registers that have to be saved and restored when invoking instrumentation can be lowered by analyzing the liveness of registers. The instrumentation code can also be exposed to the dynamic run-time system, which may perform optimizations, such as instruction trace formation, on the instrumentation code to efficiently stitch it into the application.

Figure 1 shows how instrumentation optimization can be integrated with a dynamic instrumentation toolkit (e.g., DynInst [10], Pin [19], FIST [14]). In the figure, whenever instrumentation needs to be inserted dynamically, a *base instrumenter* is invoked to determine where and how to instrument the program. The input to this base instrumenter is a sequence of binary instructions—e.g., an instruction trace—and the output is a set of unoptimized IPs. A simple intermediate representation (IR) is used to describe the unoptimized IPs; the unoptimized instrumentation code is not actually generated or inserted into the application binary.

The base instrumenter implements the instrumentation algorithm; for example, a basic block profiler would generate an IP for every block in an instruction trace. The unoptimized IPs are passed to an *instrumentation optimizer* which transforms the input IPs into a set of optimized instrumentation points. The optimizations are applied over the intermediate representation for the IPs, producing an optimized set of points. For example, using the optimizations on instrumentation to count basic blocks, a sequence of IPs can be optimized into a single IP at trace exits that covers the path from the trace entry to an exit. The intermediate representation for the optimized IPs are passed to an *instrumentation back-end* that generates and inserts the actual instrumentation code into the binary code.

3. Instrumentation Optimizations

Our goal is to apply instrumentation optimizations in many settings and preserve exact information, rather than trade precision for low overhead (e.g., sampling). The optimizations are implemented in an instrumentation optimizer that transforms IPs to improve profiling performance. The optimizations include reducing the number of run-time instances of IPs with *dynamic probe coalescing (DPC)* and reducing the overhead of an individual point with *partial context switches (PCS)* and *partial payload inlining (PPI)*.

To make the analysis needed by these optimization feasible in a dynamic setting, the optimizations operate on code regions that are straightline sequences of basic blocks with a single entry and multiple exits (i.e., instruction traces).

3.1. Dynamic Probe Coalescing

To reduce dynamic probe count, we use *dynamic probe coalescing* (DPC), which pairwise coalesces multiple probes into a single probe that does all the actions of the original probes without the loss of any information (i.e., it invokes the same payloads as the original probes, except with a one probe). The newly coalesced probe has less overhead than the original ones because the overhead of intercepting program execution is paid only once for the new probe. Many probes can potentially be combined into a single probe, which will further reduce the cost of executing the probes.

The challenge with coalescing probes is to detect the dependencies between probes and the intervening code to determine whether probes can be coalesced. For instance, the payload invoked by a probe may use the current value in a register, which could change if the probe were coalesced with some other probe. A similar dependence can arise if the payload invoked by a probe changed the state of execution in some way (e.g., a value in a register or memory). At the very least, the order of IPs often needs to be preserved. To detect these dependencies, analyses of the application code and/or payload is performed. If the analyses do not detect any dependency, the corresponding probes can be coalesced.

```
1   IPset DPC(IPset I, region r) {
2     IPset O ← ∅; IP Prev ← firstElement(I);
3     ∀i in I do {
4       ∀ instructions m betw. *Prev and *i do{
5         if ((def(i.instruction) ∩ def(m))≠∅)
6           src_reg_live ← true;
7       }
8       if (src_reg_live = false){
9         k ← coalesce(i, Prev, r);
10        O ← {(O-Prev) ∪ k}; Prev ← k;
11      }
12      else {
13        O ← {O ∪ i}; Prev ← i;
14      }
15    }
16    return O;
17  }
```

(a) Algorithm for DPC optimization

```
1
2   IP coalesce(IP i, IP j, region r) {
3     IP k ← (i . j); k.address ← j.address;
4     E ← exitBlocks(r);
5     if ∃e in E: e is an exit between i and j{
6        i.address ← e.address;
7     }
8     return k;
9   }
10
11
```

(b) Coalescing on single entry, multiple exit code

Table 1: Pseudo-code for DPC

Table 1a shows pseudo-code for the DPC optimization. In the table, the optimization takes an input set of unoptimized instrumentation points (I) and produces an optimized set of instrumentation points (O). DPC also takes an associated code region r for I. The optimization works by traversing the set of original IPs, starting from the IPs that are earliest in the code, trying to find adjacent pairs of IPs that can be combined (line 3). Consider two instrumentation points *Prev* and i, where *Prev* is earlier in the code than i. *Prev* and i can be coalesced as long as there is no intervening definition of a register used in *Prev* between *Prev* and i (lines 4-7). This condition ensures that the same information at *Prev*'s original location is also available at i. When *Prev* and i are coalesced, a new instrumentation point k is formed and inserted at i's original location (lines 8-11). k is added to O and *Prev* is removed from O (line 10). k is also marked as the new *Prev* (line 10). If *Prev* and i can not be coalesced, i is added to O and i is considered for coalescing with the next subsequent IP (line 13). In the pseudo-code, a newly coalesced point will be immediately considered for coalescing with the next subsequent IP. Instrumentation points are effectively coalesced in the downward direction and moved as late as possible. Although it is not shown, it is beneficial to coalesce in the upward direction as well.

Because the code is sequential (possibly with exits) DPC needs to consider only adjacent points because two points can be coalesced only if all intervening points can also be coalesced. For example, consider three instrumentation points: i, j, and k. Suppose i uses a register r that is defined between i and j. If i can not be combined with j, then it can not be combined with k because there is a definition of r between i and k. Note that the pseudo-code does not resolve dependencies that disable coalescing IPs. It is possible, however, to spill needed values before they are overwritten to enable coalescing. While spilling can be useful, its cost may be significant because a context switch may be needed to do the spill (e.g., for the effective address). With register liveness information, dead registers can be used to hold the spilled values to reduce the cost of making copies.

Table 1b shows how two IPs are coalesced. The IPs are initially coalesced by combining their payloads (indicated by the "." operator). For instruction traces, coalescing can combine two IPs in adjacent basic blocks. Consider an instrumentation point i in an earlier block and j in a later one. When i and j are coalesced, the newly coalesced IP is inserted in j's basic block. During execution, if the trace is exited early from i's basic block, then we must ensure that i is executed. To ensure that i is executed along an early off-trace path, instrumentation is inserted into an "exit stub block". The exit stub is a "dummy block" at a trace exit that holds the points that must be executed when the trace exits at that block. These "fix-up instrumentation points" are simply copies of IPs that were coalesced with IPs from the adjacent basic block. A fix-up instrumentation point can be an IP that resulted from a previous coalescing.

3.2. Partial Context Switch

The cost of a single IP can be lowered by applying optimizations that specialize it to the surrounding code context. We develop instrumentation optimizations to improve the instrumentation probe's performance. In particular, the cost of saving and restoring program context when calling and returning from the payload can be lowered, which is the major cost associated with a probe. This optimization is applied in static instrumentation toolkits, but it is harder for

dynamic instrumentation because information needed to reduce the context is potentially expensive to obtain.

If an IP can be placed at any instruction address, then no assumptions can be made about what registers may be live at that location without contextual knowledge. However, if the liveness of registers at an instrumentation location and in the instrumentation payload is known or can be easily computed, then the cost of saving and restoring registers when invoking the payload can be reduced. In particular, any register that is dead at an instrumentation location whether or not it is used by the instrumentation payload does not have to be saved or restored. Also, any live register at an IP that is not used by the payload does not have to be saved or restored. By analyzing register liveness, the overhead of the context switch can be significantly reduced.

Static tools like ATOM [24] can afford the cost of analysis to determine the liveness of registers to apply PCS. Instead of determining such information at run-time, another possibility is to collect the liveness information for IPs during compilation and passing that information to PCS at run-time. PCS will then generate code on-the-fly that saves and restores only those registers which are necessary.

Local register liveness can be determined relatively inexpensively on straightline code, such as basic blocks and instruction traces. This local information is valuable on some architectures that have condition code registers (e.g., SPARC and x86). Saving and restoring the condition codes can be expensive (i.e., more so than regular registers) and it is worthwhile to avoid spilling and reloading the condition codes whenever possible.

In some situations, it is also possible to spill and reload a register only once between multiple IPs. For example, in an instruction trace, registers needed by the instrumentation code can be spilled at trace entry and reloaded at the exit points when the registers used by the instrumentation are not needed (but are live) on the trace. Alternatively, registers in the instrumentation can be rewritten to use dead registers.

3.3. Partial Payload Inlining

In addition to the probe, the cost of the payload can also be reduced. Instrumentation payload often has a structure where a condition monitors program execution to decide what action to perform. For instance, a payload for a memory profiler might monitor an address range and collect values whenever an address is in that range. The payload may even have a series of conditions that guard multiple actions, where some actions may be more frequently executed than others. The frequently executed regions can be identified for *payload partial inlining (PPI)*. PPI replaces and inlines a heavyweight payload by a lightweight version that is guarded by a check. The check determines when to transfer control to the heavyweight version. PPI ensures that the inlined portion of the payload is frequently executed and reasonably small (to avoid adverse cache effects).

PPI has three steps: 1) *code partitioning*, 2) *partial inline replacement,* and 3) *run-time selection*. In the first step, the payload is partitioned into lightweight and heavyweight versions, and in the second step, IPs are optimized to use partial lining. Finally, at run-time, the guard decides which version of the payload to execute.

For code partitioning, PPI divides the payload into lightweight and heavyweight code regions. These regions are determined either by user annotations or by analysis with profile information. The lightweight region and its guard, which we collectively term a *partially inlined region* (PIR), is a small region that is frequently executed. It can be included directly in a probe to avoid a function call to the original payload. In essence, the PIR is inlined, while the heavyweight code region remains a separate function. The full payload is not inlined because it may result in too much code growth and can hurt instruction cache locality. In addition to the benefit of inlining the PIR, dividing the payload into two parts may enable PCS. Because the PIR is relatively small, it is more likely that fewer registers will have to be spilled and reloaded when invoking the inlined code, which allows PCS to be more effective.

```
1   code_region partition(code_region c) {
2     code_region pir←∅;
3     code_region cond←find_guard(c);
4     if (cond ≠ ∅) {
5       code_regions R←find_regions(cond);
6       hot←∅;
7       ∀r in R do { // find small, hot region
8         if ((hot=∅ || (hotness(r)>HOT_THRES&&
9             hotness(r) > hotness(hot_region))
10            && size(r) < SIZE_THRES)
11          hot←r;
12      }
13      if (hot≠∅) {
14        remove(hot, c); // remove hot region
15        remove(cond, c); // remove condition
16        pir←gen_inline_region(cond, hot, c);
17      }
18    }
19    return pir;
20  }
```

Table 2: Pseudo-code for extracting partial inlined code region

Table 2 shows a simplified algorithm for partitioning the payload into the partially inlined region and heavyweight regions. The algorithm takes as an input a code region *c* annotated with edge weights (from a profile) to be partitioned. The algorithm looks for a hot region in *c* that is guarded by a condition at the entry of *c* and has an exit from *c*. The algorithm tries to identify the guard condition (*cond*) for the inlined code (line 3), which should be at the entry of *c*. If a suitable condition is found, then *c* is partitioned into several hot regions, *R*, starting from *cond* (line 5). The regions in *R* are single entry and multiple exit (e.g., an instruction trace), with the entry being a successor to the guard condition. To simplify the algorithm, all exits from the regions in *R* should exit the surrounding region *c* (this restriction can be relaxed, where only an exit on the hot path in *R* should exit *c*). The regions are processed to find the hottest one that meets a constraint on code size (lines 6-12). The region size must be below the threshold size to avoid inlining too much code. If a small, hot region *hot* is found, then that region and its guard are removed from *c* to form the partial inline region, *pir* (lines 13-17), which will be inlined. The heavyweight region is *c* after *hot* has been removed; a call is inserted in the guard to invoke *c* when *pir's* condition fails.

In the second step of PPI, IPs are optimized based on the PIR. This step inlines the PIR into an instrumentation probe. IPs that invoke different (or multiple) payloads can be handled by inlining the appropriate PIR for each payload. During run-time, the final step occurs, where the guard is executed to call the partial inlined or heavyweight regions.

4. INS-OP

We implemented an instrumentation optimizer, called INS-OP, that is integrated with a dynamic binary translator, Strata [23], and a dynamic instrumentation system, FIST

[14]. INS-OP, Strata, and FIST serve as a "dynamic instrumentation toolkit" that can be used to implement new instrumentation algorithms and optimizations.

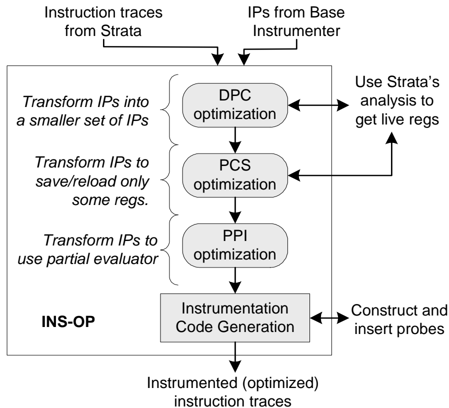

Figure 2: Instrumentation Optimizer

INS-OP implements DPC, PPI and PCS as passes as shown in Figure 2. INS-OP accepts an instruction trace (formed online with the NET algorithm [2]) from Strata and a set of IPs from the base instrumenter. The IPs are represented as annotations in the control flow graph for the code region. INS-OP will work with any base instrumenter that generates IPs in INS-OP's intermediate representation. The optimizer uses DPC to reduce the unoptimized IPs. After DPC, the cost of an individual IP is tackled. PCS replaces the full context switch with a partial one that saves and reloads the registers used by the payload. Then, PPI does partial inlining of the payload. The optimized IPs and the input code region are passed to the FIST instrumentation code generator [14]. FIST constructs and inserts the code for each IP. Finally, FIST's output is passed to Strata.

5. Case Studies of Using INS-OP

To explore the usefulness of instrumentation optimization, we applied INS-OP to two case studies: one on program profiling and the other for simulating processor caches. The profilers insert limited instrumentation and have relatively inexpensive payloads. The cache simulator, on the other hand, has instrumentation that is relatively expensive, with many instrumentation points. Hence, these case studies represent a spectrum for exploring the effectiveness of INS-OP.

All experiments were done on a 500 MHz Sun Blade with 256 MB of RAM and Solaris 9. The benchmark suite was SPECint2K, using gcc 2.95.3 with flags "-O3 -mv8". The train data set was used for profiling and the reference set for evaluation. The results include all run-time costs.

For the profiler case study, the benchmarks were run for a maximum of one billion profile values. For the cache simulation study, the benchmarks were run to completion. The simulated cache is a split instruction and data two-way set associative L1 cache with 16 KB storage and 32 byte lines.

5.1. Case Study: Program profiling

To investigate how INS-OP performs for profiling, we conducted a case study with several program profilers as shown in Table 3. Although these profilers are similar in some respects, they represent a spectrum of commonly used profilers that instrument at different granularities.

Profilers	Description	Base	Opt.	Speedup
BB count	Execution count of each basic block	487s	218s	2.15x
Path profile	Gather blocks executed along a path	448s	218s	1.96x
Address profile	Collect load/store addresses	589s	219s	2.54x
Value profile	Collect values used by loads and stores	585s	211s	2.63x
Branch history	Record taken and not-taken branch history	511s	212s	2.26x
Call-chain	Record order of function calls and returns	712s	585s	1.26x

Table 3: Speedup due to nstrumentation optimization

Table 3 shows average run-time for each profiler when run on the SPECint2K benchmarks, with no instrumentation optimization ("base") and with instrumentation optimization ("Opt."). The numbers are the average run-times in seconds for the benchmarks. For example, the first row shows a basic block count profiler in which the average run-time is reduced from 487 seconds to 218 seconds. INS-OP improved profiling performance by 2.15 times in this case.

The table shows that the speedup varies from 1.26 for call-chain profiling to 2.63 for value profiling. The difference in speedup is related to the number of IPs and payload cost. For example, the address profiler gathers all memory address generated during program execution. In comparison to the other profilers, it has a high number of IPs and a more expensive payload, which makes the optimizations more effective. The call-chain profiler, on the other hand, instruments only calls and returns. That is, the call-chain profiler has sparse IPs, which reduces the effectiveness of some optimizations (e.g., DPC). Thus, the speedup is smaller (1.26), but INS-OP is still effective due to PPI and PCS.

From these results, we conclude that INS-OP is effective and can reduce instrumentation overhead. INS-OP also eased the burden of implementing these profilers. It took only a day to develop the profilers because the harder task of minimizing instrumentation cost was left to INS-OP. Instead, we only needed to describe what instructions to instrument, what information to extract, and how to use it.

5.2. Case Study: Cache Simulation

In this study, the instrumentation application is a direct-execution processor simulator. These simulators, including SimOS/Embra [26] and Shade [4], instrument the executing program to dynamically model a computer architecture. One way to model caches in these simulators is to instrument all memory operations to collect a trace of instruction and data addresses. Profiling all memory operations—i.e., all instruction fetches and data load and stores—is expensive because every instruction has to be instrumented. Instead of instrumenting all instructions, the cost and amount of instrumentation can be optimized by INS-OP.

We integrated an instrumenter for memory profiling into INS-OP. The base instrumenter determines a set of IPs that collect data and instruction address streams. To extract the address streams, the base instrumenter generates two types of instrumentation points: data and instruction points. The *data points* are generated for every load and store and the *instruction points* are generated for every instruction. Probes for data points determine source values for effective addresses, which are passed to the instrumentation payload.

Similarly, the probes for instruction points determine the instruction address from the original binary and pass the address to the payload. The set of unoptimized IPs from the base instrumenter are given to INS-OP for optimization.

To model processor caches, the payload is a cache simulator. For the cache simulator, we used a simulator based on Embra [26]. Embra uses a table lookup to quickly detect cache hits at the expense of handling misses, and because cache hits are more likely than misses, it is able to achieve fast performance [26]. In the original Embra tool, the hit handling code is inlined at an instrumentation site [26]. In our case, the base cache simulator combines the hit and miss handler into a single payload. When PPI is used, the hit handler is inlined automatically by INS-OP. Embra avoids context switches when detecting hits by hard-coding reserved registers for simulation [26]. In INS-OP, PCS is applied automatically on the hit detector after PPI to get the same effect. Embra does not apply DPC.

Program	Native (Sec)	Strata-Embra	Strata-Embra-Opt
mcf	1,813	5.5x	2.5x
twolf	3,534	15.7x	7.2x
gcc	1,364	25.2x	11.1x
vpr	831	20.4x	10.1x
parser	1,979	21.6x	8.8x
vortex	2,747	21.5x	10.6x
gzip	1,192	34x	13.1x
bzip	1,325	26.7x	8.1x

Table 4: Slowdown of cache simulators over native execution

We compared INS-OP for cache simulation relative to Embra as shown in Table 4. The table shows the native execution time, the slowdown of Strata-Embra and Strata-Embra-Opt. Strata-Embra is Witchel and Rosenblum's original simulator [26] implemented in our framework. Strata-Embra-Opt is Strata-Embra optimized by INS-OP.

As the table shows, Strata-Embra has a slowdown of 5.5–34x, with an average of 21x. When Strata-Embra-Opt is considered, it is 2.5–13x slower than native execution, with an average 8.9x slower. Strata-Embra-Opt is 2–3.3x faster (average 2.4x) than Strata-Embra due to instrumentation optimizations. These results are encouraging because INS-OP is a general instrumentation optimization framework that can be applied in different settings. Tools like Embra are not general systems—they are designed for a specific purpose. INS-OP is more general, yet it achieves better performance.

6. Related Work

Instrumentation has been used for a number of purposes, including program profiling [5,15,19,20,24], dynamic optimization [1,2], software security [12], and binary translation [6,7,8]. These systems use instrumentation to monitor and gather information about a program. However, unlike INS-OP, they do not use instrumentation optimizations.

Systems like Dyninst [10] and Paradyn [17] use instrumentation probes based on fast breakpoints to keep instrumentation overhead low. Both Dyninst and Paradyn were built for dynamic instrumentation, and to the best of our knowledge, their instrumentation techniques were not designed for instrumentation optimization. FIT is a ATOM-like static system that focuses on retargetability rather than instrumentation optimization [5]. PIN is a dynamic instrumentation framework [19]. It has support for writing custom instrumentation optimizations, such as allowing instrumentation on instruction traces. However, these optimizations have to be written and applied by the programmer.

There are many tools for cache simulation that use direct execution. FastSim [21] is a processor simulator that uses static instrumentation. It does not apply instrumentation optimization, but uses memoization to improve simulation performance. INS-OP could be integrated with FastSim to improve instrumentation performance. Similarly, Embra [26] uses dynamic binary translation and inlines instrumentation in the binary code. Embra could benefit from the DPC optimization. Fast-Cache [16] is a fast data cache simulator that uses static instrumentation for every memory instruction (load and store) and calls a cache simulator when there is a miss. Fast-Cache could benefit from both DPC and PCS.

7. Conclusion

This paper described an approach to reduce dynamic instrumentation overhead with "instrumentation optimizations". These optimizations reduce the number of run-time instances of an instrumentation point and the cost of individual instrumentation points with dynamic probe coalescing, partial payload inlining, and partial context switches. We presented an optimizer, INS-OP, for dynamic instrumentation optimization and described two case studies that evaluated the effectiveness of instrumentation optimizations for profiling and processor cache simulation. INS-OP improved profiling performance by 1.26–2.63x and cache simulation by 2–3.3x. The instrumentation optimizations are general and can be used in other applications, such as checking program invariants, alias analysis, and dynamic slicing.

8. References

[1] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code", *Conf. on Programming Language Design and Implementation*, 2001.

[2] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent dynamic optimization system", *Conf. on Prog. Language Design and Implementation*, 2000.

[3] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", *Int'l. Symp. on Code Generation and Optimization*, 2003.

[4] R. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling", TR# 93–06–06, Computer Science, Univ. of Washington, June 1993.

[5] B. De Bus, D. Chanet, B. de Sutter, L. Van Put, and K. de Bosschere, "The design and implementation of FIT: A flexible instrumentation tool", *Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[6] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges", *Int'l. Symp. on Code Optimization and Generation*, 2003.

[7] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. Fisher, "DELI: A new runtime control point", *Int'l. Symp. on Microarchitecture*, 2002.

[8] K. Ebcioglu and E. Altman, "DAISY: Dynamic compilation for 100% architectural compatibility", *Int'l. Symposium on Computer Architecture*, 1997.

[9] M. Ernst, J. Cockrell, W. Griswold, D. Notkin, "Dynamically discovering likely program invariants to support program evolution", *IEEE Trans. on Software Engineering*, 27(2), 2001.

[10] J. Hollingsworth, B. Miller, M. Goncalves, et al., "MDL: A language and compiler for dynamic program instrumentation", *Parallel Arch. and Compilation Techniques*, 1997.

[11] P. Kessler, "Fast breakpoints: Design and implementation", *Conf. on Prog. Language Design and Implementation*, 1990.

[12] V. Kiriansky, D. Bruening and S. Amarasinghe, "Secure execution via program shepherding", *Security Symposium*, 2002.

[13] N. Kumar, B. R. Childers, D. Williams, J. W. Davidson, and M. L. Soffa, "Compile-time planning for software dynamic translation", *Int'l. Journal on Parallel Programming*, 2005.

[14] N. Kumar, J. Misurda, B. Childers, and M. L. Soffa, "Instrumentation in software dynamic translators for self-managed systems", *Workshop on Self-Managing Systems*, 2004.

[15] J. R. Larus and E. Schnarr, "EEL: Machine-independent executable editing", *Conf. on Prog. Language Design and Implementation*, 1995.

[16] A. R. Lebeck and D. A. Wood, "Active memory: A New Abstraction for Memory System Simulation", *ACM SIGMETRICS*, pp. 220-230, 1995.

[17] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, et al., "The Paradyn parallel performance measurement tools", *IEEE Computer*, 28(11), Nov. 1995.

[18] J. Misurda, J. Clause, J. L. Reed, B. R. Childers and M. L. Soffa, "Demand-driven structural testing with dynamic instrumentation", *Int'l. Conf. on Software Engineering*, 2005.

[19] Pin, Program instrumentation, http://rogue.colorado.edu/Pin

[20] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad, "Instrumentation and optimization of Win32/Intel executables using Etch", *Windows NT Workshop*, 1997.

[21] E. Schnarr, "Fast Out-of-Order Processor Simulation Using Memoization", *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.

[22] K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation", *Annual Computer Security Applications Conference*, 2002.

[23] K. Scott, N. Kumar, S. Veluswamy, B. Childers, J. Davidson, M. L. Soffa, "Reconfigurable and retargetable software dynamic translation", *Int'l. Symp. on Code Generation and Optimization*, 2003.

[24] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools", *Conf. on Prog. Design and Implementation*, 1994.

[25] M. Tikir and J. Hollingsworth, "Efficient instrumentation for code coverage testing", *Int'l. Symp. on Software Testing and Analysis*, 2002.

[26] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation", *Conf. on Measurement and Modeling of Computer Systems*, 1996.

[27] Q. Wu, A. Pyatakov, A. N. Spiridonov, et al., "Exposing Memory Access Regularities using Object-Relative Memory Profiling", *Int'l. Symp. on Code Generation,* 2004.

[28] X. Zhang and R. Gupta, "Cost effective dynamic program slicing", *Conf. on Prog. Language Design and Implementation*, 2004.