Bursty Tracing: A Framework for Low-Overhead Temporal Profiling

Martin Hirzel
Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
`hirzel@colorado.edu`

Trishul Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
`trishulc@microsoft.com`

Abstract

With processor speed increasing much more rapidly than memory access speed, memory system optimizations have the potential to significantly improve program performance. Unfortunately, cache-level optimizations often require detailed temporal information about a program's references to be effective. Traditional techniques for obtaining this information are too expensive to be practical in an on-line setting. We address this problem by describing and evaluating a framework for low-overhead temporal profiling. Our framework extends the Arnold-Ryder framework that uses instrumentation and counter-based sampling to collect frequency profiles with low overhead. Our framework samples bursts (sub-sequences) of the trace of all runtime events to construct a temporal program profile. Our bursty tracing profiler is built using Vulcan, an executable-editing tool for x86, and we evaluate it on optimized x86 binaries. Like the Arnold-Ryder framework, we have the advantages of not requiring operating system or hardware support and being deterministic. Unlike them, we are not limited to capturing temporal relationships on intraprocedural acyclic paths since our trace bursts can span procedure boundaries. In addition, our framework does not require access to program source or recompilation. A direct implementation of our extensions to the Arnold-Ryder framework results in profiling overhead of 6-35%. We describe techniques that reduce this overhead to 3-18%, making it suitable for use in an on-line setting.

1 Introduction

Dynamic optimization can use profile information from the current execution of a program to decide what and how to optimize. That is an advantage over static and even feedback-directed optimization. On the other hand, dynamic optimization must be more concerned with the profiling overhead, since the slow-down from profiling has to be recovered by the speed-up from optimization. A common way to reduce the overhead of profiling is *sampling*: instead of recording all the information that may be useful for optimization, sample a small, but representative fraction of it.

Sampling typically counts the frequency of individual events such as calls or loads [3]. However, many dynamic optimizations exploit causality between two or more events. For example, prefetching with Markov-predictors uses pairs of data accesses [16]; some recent transparent native code optimizers focus on single-entry, multiple-exit code regions [6, 13]; cache-conscious data placement during generational garbage collection lays out sequences of data objects [11, 10]. For lack of low-overhead temporal profilers, these systems usually employ event profilers. But as Ball and Larus point out, event (node or edge) profiling may misidentify frequencies of event sequences [7].

The sequence of all events during execution is the *trace*; a *burst* is a subsequence of the trace. Arnold and Ryder present a framework that samples bursts [4]. In their framework, the code of each procedure is duplicated (see Figure 1). Both versions of the code contain the original instructions, but only one version is instrumented to also collect profile information. The other version only contains *checks* at procedure entries and loop back-edges that decrement a counter $nCheck$, which is initialized to $nCheck_0$. Most of the time, the checking code is executed. Only when $nCheck$ reaches zero, a single intraprocedural acyclic path of the instrumented code is executed and $nCheck$ is reset to $nCheck_0$.

The Arnold-Ryder framework has several advantages: it is implemented without hardware or operating system support; it is deterministic; its overhead is easy to control. Let t_{checking} and $t_{\text{instrumented}}$ be the running times if we always stay in one version of the code. Then the running time of the program is

$$t(nCheck_0) = \frac{nCheck_0 \cdot t_{\text{checking}} + 1 \cdot t_{\text{instrumented}}}{nCheck_0 + 1} \quad (1)$$

Let t_{orig} be the running time of the original code. The overhead is $t(nCheck_0)/t_{\text{orig}} - 1$, which approaches $t_{\text{checking}}/t_{\text{orig}} - 1$ for large $nCheck_0$. We call $t_{\text{checking}}/t_{\text{orig}} - 1$ the *basic overhead*.

A limitation of the Arnold-Ryder framework is that it stays in the instrumented code only for the time between
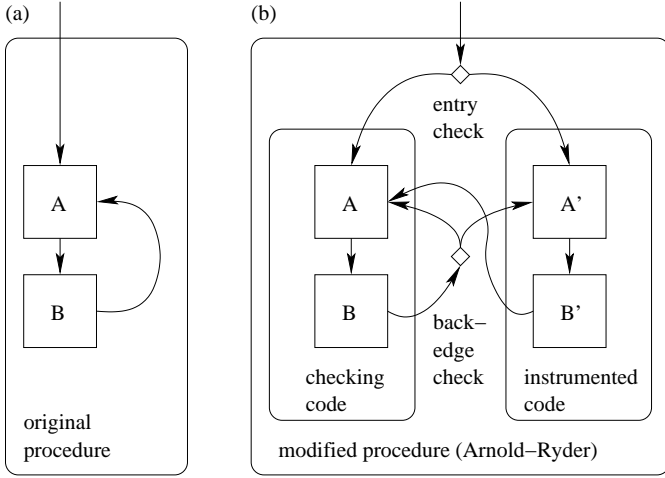
1

Figure 1: Arnold-Ryder framework.

two checks. Since it has checks at every procedure entry and loop back-edge, a burst captures only one acyclic intraprocedural path's worth of trace. Consider for example the code fragment:

for$(i = 0; i < n; i++)$
    if$(\ldots)$ $f()$;
    else $g()$;

The Arnold-Ryder framework would be unable to distinguish the traces $fgfgfgfg$ and $ffffgggg$. But for example in a dynamic optimizer based on optimizing single-entry multiple-exit regions this information may make the difference between executing optimized code most of the time or not.

This paper describes our enhancements to the Arnold-Ryder framework.

- We extend the Arnold-Ryder framework to sample longer bursts (Section 2), making it more useful for temporal profiling. (Arnold and Ryder actually mention this possibility, but do not report trying it out.)

- We report experiences with our extended framework on x86 binaries (Section 3). Arnold and Ryder had implemented their framework in the JIT compiler of the Jikes RVM[1] for Java class files [1], we have implemented it with Vulcan for x86 binaries [20]. Among other things, this gives us language-independence.

- We reduce the basic overhead (Section 4). Out of the box, our extended Arnold-Ryder framework has a basic overhead of 6-35% in our setting, we reduced it to 3-18% by eliminating checks. We eliminate redundant checks on procedure entries by analyzing recursive cycles, and eliminate checks from loops that do not yield interesting profile information.

---

[1]The Jikes RVM is an open source research virtual machine for Java that was formerly called Jalapeño.

2   Profiling Longer Bursts

A burst of the Arnold-Ryder framework begins at a check (procedure entry or loop back-edge) and extends to the next check: it captures one intraprocedural acyclic path. We extended the framework so that bursts can extend over multiple checks, possibly crossing procedure boundaries. This way, we obtain interprocedural, context-sensitive, and flow-sensitive profiling information.

Our extension is to make the Arnold-Ryder framework symmetric (see Figure 2). While in the checking code, we decrement $nCheck$ at every check. When it reaches zero, we initialize $nInstr$ with $nInstr_0$ (where $nInstr_0 \ll nCheck_0$) and go to the instrumented code. While in the instrumented code, we decrement $nInstr$ at every check. When it reaches zero, we initialize $nCheck$ with $nCheck_0$ and go to the checking code. Figure 3 shows the logic of a check. Note that in the common case (we were in the checking code and stay there), the check is basically the same as in the original Arnold-Ryder framework. Comparing the common case of Figure 3 in this paper to Figure 3 in [4], we see that the fast path consists of a decrement and a conditional branch in both cases.
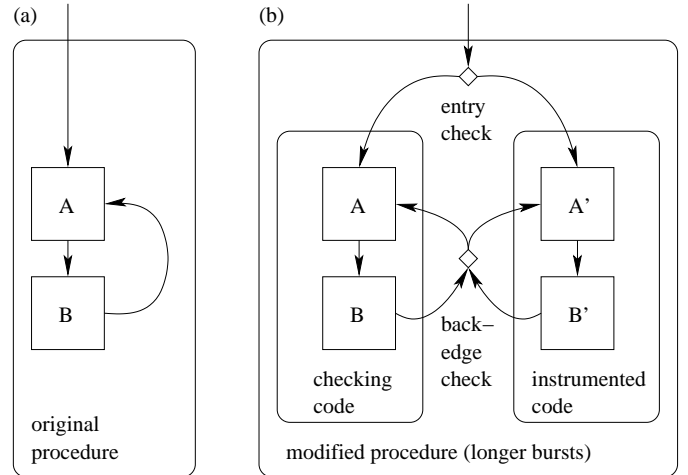

Figure 2: Long-burst extended Arnold-Ryder framework.

The extended framework still does not require any hardware or operating system support, it is still deterministic, and the profiling overhead is still easy to control. Equation 1 becomes

$$t(nCheck_0, nInstr_0) = \frac{nCheck_0 \cdot t_{\text{checking}} + nInstr_0 \cdot t_{\text{instrumented}}}{nCheck_0 + nInstr_0} \quad (2)$$

The overhead depends on the *sampling rate* $r = nInstr_0/(nCheck_0 + nInstr_0)$. For a low sampling rate r, the overhead approaches the basic overhead $t_{\text{checking}}/t_{\text{orig}} - 1$.

Our extended framework can accidentally transition to the wrong version of the code on a procedure return. Let

```
// fast path, executed on every check
nCheck−−;
if(nCheck ≠ 0) goto target_checking;            // we were in checking code and stay there
// fall-through to remainder of check, executed infrequently
nCheck = 1;
if(nInstr == 0){
    nInstr = nInstr_0;
    goto target_instrumented;                    // transition from checking to instrumented code
}
nInstr−−;
if(nInstr ≠ 0) goto target_instrumented;        // we were in instrumented code and stay there
nCheck = nCheck_0;
goto target_checking;                            // transition from instrumented to checking code
```

Figure 3: Logic of a check.

us say the checking code of f calls g; it pushes a return address in the checking code on the stack. Let us say g does a regular transition to the instrumented code and then returns to f. We have accidentally transitioned from the instrumented code to the checking code. Note that the mistake will be corrected at the next check, and that we have at most one accidental transition per return after the regular transition. This situation could theoretically lead to irregularities in how many profile events there are per burst, and it could make the profiling overhead deviate from Equation 2. We measured both and found that returns to the wrong version of the code were not a problem in practice.

3   Profiling x86 Binaries

Arnold and Ryder implemented their framework with the Jikes RVM [1]. They compile Java class files to PowerPC and insert the checks at compile-time. We implemented our version of the framework with Vulcan, an executable editing tool for x86 [20]. We compile C and C++ programs to x86 and insert the checks into the binary (at post-link time). Our framework has the advantage of requiring access only to the executable but suffers from the disadvantage that the compiler cannot optimize the checks.

Arnold and Ryder observe basic overheads from 0.5-8.4%, the average over all benchmarks is 3.6%. Our implementation of the extended Arnold-Ryder framework yields basic overheads from 5.6-35.3% (see Figure 6), the average over all benchmarks is 15.9%. In their setting, the basic overhead seems low enough for dynamic optimizations, whereas in our setting, it is too high.
There are at least two reasons for this discrepancy. First, Java programs already have higher overhead from the runtime system and from the larger semantic gap that the compiler must bridge. The overhead of the checks is smaller when compared to an already slower program.

Second, in Arnold and Ryder's setting, the JIT compiler does register allocation and instruction scheduling for the checks.

Our experiences show that the extended Arnold-Ryder framework is general enough for use in various settings, but in our setting, its basic overhead is too high for dynamic optimization.

4   Decreasing the Overhead: Fewer Checks

As discussed in the previous section the basic overhead of our extended Arnold-Ryder framework is too high for dynamic optimization. The framework has checks at all procedure entries and loop back-edges to insure that the program can never loop or recurse for an unbounded amount of time without executing a check. If this were not the case, sampling could miss too much profiling information (when the program spends an unbounded amount of time in the checking code), or the overhead could become too high (when the program spends an unbounded amount of time in the instrumented code).

We want to reduce the basic overhead by executing fewer checks, but still guarantee that the program never spends an unbounded amount of time without executing a check. Note that thread-yield points, gc-safe points, and asynchronous-exception points have similar requirements. For all of these, we want to execute as few checks as possible to reduce overhead, but must guarantee that the time between checks is bounded. Little is said about these problems in the literature. The rest of this section discusses techniques for eliminating checks at procedure entries and loop back-edges.

3

4.1 Eliminating Checks at Procedure Entries

Figure 4 shows a static *call-graph*: a directed graph $G = (N, E)$ where the nodes N are procedures and there is an edge $(f, g) \in E$ iff the code of f contains at least one direct call to g. We want to insert checks on procedure entries so that the program cannot recurse for an unbounded amount of time without executing a check. In other words, we want to find a minimum set $C \subseteq N$ of nodes such that every cycle in the graph contains at least one of them. This problem is NP-hard (you can reduce VERTEX-COVER to it).
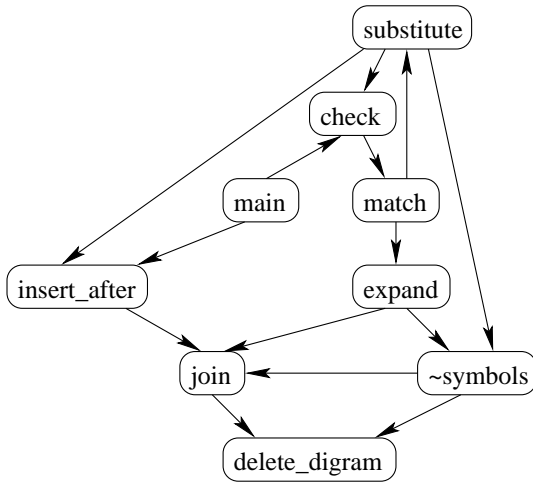

Figure 4: A call-graph.

We approximate a good set $C \subseteq N$ of procedures f to put entry-checks on:

$$C = \Big\{ f \in N \mid \neg \textit{is\_leaf}(f) \wedge \big(\textit{is\_root}(f) \vee$$
$$\textit{addr\_taken}(f) \vee$$
$$\textit{recursion\_from\_below}(f) \big) \Big\} \tag{3}$$

You never need a check on entry to a leaf procedure (a procedure that calls nothing), since it cannot be part of a recursive cycle. Otherwise, we put a check on entry to every root (a procedure that is only called from the outside world) to make sure we start off in the right version of the code; we put a check on entry to every procedure whose address is taken, since it may be part of recursion with indirect calls; and we put a check on entry to every procedure with recursion from below. A procedure f has recursion from below iff it is called by a procedure g in the same strongly connected component as f that is at least as far away from the roots. The distance of a procedure f from the roots is the minimum length of the shortest path from a root to f.

Consider for example Figure 5 and assume there are no indirect calls. The only root is *main*, the only

leaf is *delete_digram*. With a breadth-first search, we find the distances from the root to the other nodes. The only non-trivial strongly connected component is $\{check, match, substitute\}$. Only the procedure *check* has recursion from below, since it is called by *substitute* which is further away from *main*. If there is a check on entry to every procedure in $C = \{main, check\}$, the program cannot recurse indefinitely without executing checks.
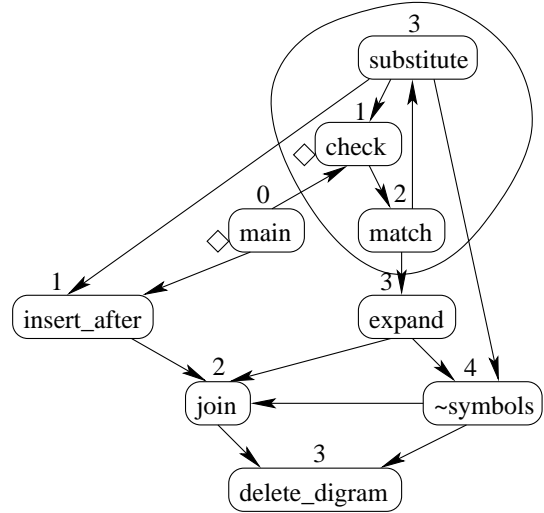

Figure 5: An analyzed call-graph.

The *recursion_from_below* heuristic guarantees that there is no recursive cycle without a check and breaks the ties to determine where in the cycle to put the check (similarly to back-edges in loops). We break ties so that checks are as far up in the call-stack as possible. This should reduce the number of dynamic checks.

4.2 Eliminating Checks at Loop Back-Edges

Static optimizers do a good job for tight inner loops. A dynamic optimizer that tries to complement a static optimizer may often find the profiling information from tight inner loops of little interest. At the same time, checks at the back-edges of tight inner loops can become extremely expensive. For example, when you prefetch based on hot data streams [8], loops that compare or copy arrays should not have checks. They are easy to optimize statically, the check on the back-edge is almost as expensive as the loop body, and the loop body contains too little work to overlap with the prefetch.

We define *k-boring loops* as loops with no calls and at most k profiling events of interest. We do not instrument either version of the code of a k-boring loop, and we do not put a check on its back-edge. This way, we never spend an unbounded amount of time in instrumented code. We may spend an unbounded amount of time in uninstrumented

4

code without executing a check, but if the k-boring hypothesis holds we do not miss interesting profiling information. In some of our experiments, the quality of the profile actually improved when we skipped 4-boring loops: we measured the quality of a profile by its ability to detect hot data streams, and eliminating k-boring loops helps sampling more interesting events.

We considered other techniques for eliminating checks on loop back-edges. If you can prove that a loop has only a small, fixed number of iterations, it does not need a check on the back-edge. Likewise, if you can prove that a loop body will always execute a check, the loop does not need a check on the back-edge. Another technique would be to combine the loop counter with the profiling counter; if they are linearly related, you do not need to update the profiling counter and can still execute checks via a predicate on the loop counter. Because they are hard to implement on binary code and their benefits are unclear, we did not implement any of these ideas.

5 Experimental Evaluation

This section answers three questions: How much overhead does our profiling framework incur? What is the quality of temporal profiles collected with our bursty tracing technique, and how does this relate to the sampling rate and length of bursts? How do our techniques for reducing framework overhead affect the quality of profiles?

5.1 Overhead

We measured the effectiveness of our techniques for reducing the overhead of the extended Arnold-Ryder framework on a 1000MHz Pentium III with 512MB of RAM. The benchmarks are x86 binaries compiled with full optimization; note that procedure inlining and loop unrolling also reduces the number of checks. We did every run three times and took the geometric mean of the running times.

The profiling overhead depends on the basic overhead and the sampling rate. Let t_{orig} be the running time of the original code and t_{checking} and $t_{\mathrm{instrumented}}$ be the running times if we always stay in one version of the code. Then given a sampling rate $r = nInstr_0/(nCheck_0 + nInstr_0)$ our framework has the overhead

$$overhead(r) = basic\_overhead + r \cdot \frac{t_{\mathrm{instrumented}} - t_{\mathrm{checking}}}{t_{\mathrm{orig}}}$$
$$(4)$$

The basic overhead is due to the checks used to implement the profiling framework. The other component is mainly due to the instrumentation doing the actual profiling and scales linearly with the sampling rate. Since we are interested in the overhead of the general profiling framework as opposed to the overhead of the specific profiling task, we present numbers for the basic overhead only.

Figure 6 shows the basic overhead $t_{\mathrm{checking}}/t_{\mathrm{orig}} - 1$, where t_{checking} is obtained by setting $nCheck_0 = \infty$ and $nInstr_0 = 1$, and t_{orig} is the running time of the original, unmodified binary.

The "orig" bar of each group shows the basic overhead when we have a check at every procedure entry and loop back-edge. Bars EL,EC,EN show how much removing procedure entry checks reduces the overhead; note that eliminating all entry checks is not a practical alternative, but gives a limit for how much can be gained by eliminating entry checks. Bars L4,L10,LN show how much removing loop back-edge checks reduces the overhead; note that eliminating all back-edge checks is not a practical alternative, but gives a limit for how much can be gained by eliminating back-edge checks. The last bar EC+L4 shows the remaining basic overhead for our preferred version of the extended Arnold-Ryder framework.

The numbers show that our techniques significantly reduce the basic overhead of our extended Arnold-Ryder framework. With the exception of 305.espresso, the basic overhead seems low enough for dynamic optimizations.

5.2 Case Study: Hot Data Streams

Being based on instrumentation, our profiling framework is flexible enough to support various different profiling tasks. We used it for finding hot data stream profiles [8]. This Section describes hot data stream profiles and a way to compare them.

For hot data stream profiling, the instrumented code records *data references* (dynamic executions of loads or stores). A *data stream* is a subsequence of the data reference trace that exhibits regularity. If the regularity magnitude of a data stream exceeds a predetermined threshold, it is called a *hot data stream*. The regularity magnitude of a data stream is the part of the trace it accounts for. Given a data stream v, the regularity magnitude is $v.heat = v.length * v.frequency$, where $v.frequency$ is the number of non-overlapping occurrences of v in the trace. We set the heat threshold such that all hot data streams together account for 90% of all traced data references.

A hot data stream *profile* is a set of hot data streams and their regularity magnitudes. We measure the quality of a profile by computing its overlap with the profile of the trace of all data references [9]. Let P and Q be two hot data stream profiles. The overlap for each hot data stream is the minimum of its contribution to P and Q, possibly zero if it appears only in one profile. For example, if $v.heat_P = 3\%$ and $v.heat_Q = 4\%$, then $overlap(v, P, Q) = 3\%$. The overlap of profiles P and Q is the sum of the overlaps for all hot data streams, i.e.

$$overlap(P, Q) = \sum_{v \in P \cup Q} \min\{v.heat_P, v.heat_Q\} \quad (5)$$

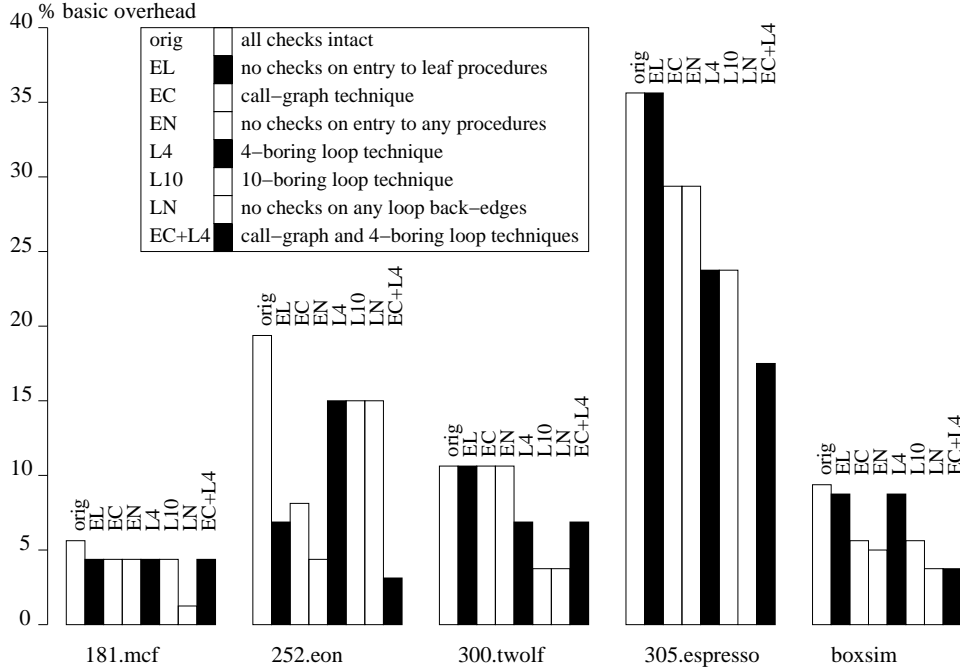Since object addresses will change from one run to another,

Figure 6: Basic overhead for various overhead reduction techniques.

we consider two hot data streams to be the same if they have the same *access signature*, which is the ordered list of instruction PCs that generated the reference sequence. This methodology follows Chilimbi's paper on the stability of temporal data reference profiles [9]. Our overlap corresponds to "exact overlap" in that paper. Note that because we want long streams for optimization, we measure overlap on streams of length 10-40, whereas [9] measures overlap on streams of length 2-100.

5.3 Profile Quality

We evaluate how longer bursts improve the quality of temporal profiles using the profiling task of detecting hot data streams as an example (see Section 5.2).

Figure 7 shows the overlap of the hot data stream profiles found by sampling (using different $nCheck_0{:}nInstr_0$) with the hot data stream profile found by collecting the complete trace of all data references. The higher this overlap, the higher the quality of the sampled profile. The bars can be grouped by the sampling rate, which is proportional to the overhead in Equation (4): bars $20{:}1, 200{:}10, 1000{:}50$ correspond to a 5% sampling rate, bars $100{:}1, 1000{:}10, 5000{:}50$ correspond to a 1% sampling rate, and bars $200{:}1, 2000{:}10$ correspond to a 0.5% sampling rate. For the bars $20{:}1, 100{:}1, 200{:}1$ we sample only bursts of length one ($nInstr_0 = 1$), which yields profiles corresponding to the original Arnold-Ryder framework but for the eliminated checks.

Figure 7 shows that when the bursts are longer, the quality

of the profile improves given a fixed allowance of overhead. In some cases, longer bursts even improve the profile quality over profiles obtained with higher overhead, but shorter bursts.

Figure 8 shows the impact of our overhead reduction techniques on the profile quality. It shows the overlap of the hot data stream profile found by sampling using $nCheck_0{:}nInstr_0 = 1000{:}50$ with the hot data stream profile found by collecting the complete trace of all data references.

The "orig" bar of each group shows the overlap when we have a check at every procedure entry and loop back-edge. Bars EL,EC,EN show how removing procedure entry checks affects the overhead; note that eliminating all entry checks is not a practical alternative, but gives a limit for how much impact eliminating entry checks can have. Bars L4,L10,LN show how much removing loop back-edge checks affects the overhead; note that eliminating all back-edge checks is not a practical alternative, but gives a limit for how much impact eliminating back-edge checks can have. The last bar EC+L4 shows the overlap for our preferred version of the extended Arnold-Ryder framework.

Comparing bars "orig" and EC+L4, we see that except for 305.espresso, our preferred settings either have no effect on accuracy or improve accuracy. By eliminating profiling in certain portions of the program, we spend more time sampling the more interesting parts. This simultaneously reduces noise and gives more information about the interesting portions of the program, leading to a higher quality profile.
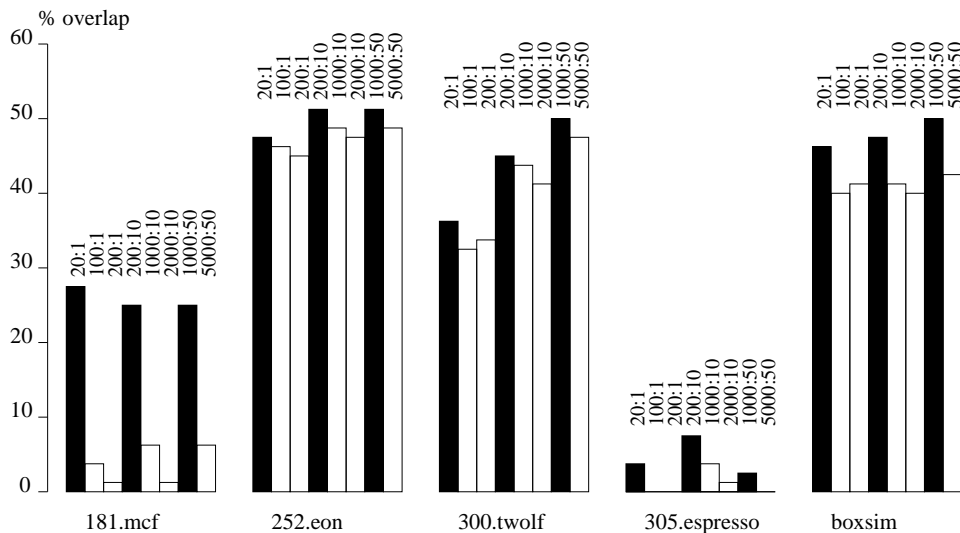
6

Figure 7: Overlap of hot data streams when sampling with different $nCheck_0 : nInstr_0$.

6 Related Work

The goal of our profiling framework is to work with low overhead and yet find causality in the profile. In this section, we review both low-overhead profilers and profilers that discover causality of events instead of just counting events independently.

6.1 Low-overhead Profilers

One way to reduce the overhead of profiling is to do most of the work in hardware, possibly in parallel to the main computation. Techniques range from configurable performance counters that can trigger software interrupts [3, 12] over selection and compression [19] to programmable co-processors [15, 23]. Hardware profilers are typically instruction-oriented: they count instructions with certain characteristics. The more sophisticated the hardware profiler, the less work remains to be done in software, so that a wide variety of profiling tasks can be done with low overhead. The draw-back is that stock hardware does not yet have sophisticated profiling support built in.

Another way to reduce the overhead of profiling is compression. It is safe for profile compression to lose information, but the results must be a summary of the original data that allows immediate and fast extraction of the relevant information. Compression techniques that fit this bill range from counting, hashing, and other hardware preprocessing (e.g. [19]) to induction of hierarchical grammars [17].

Finally, a common way to reduce the overhead of profiling is sampling. Hardware sampling is typically counter-based. In DCPI and ProfileMe, a sample is taken via an interrupt when one of the hardware performance counters of the Alpha processor overflows [3, 12]. In stratified sampling, events are hashed and hits for each hash bucket are counted; when the counter for a hash bucket overflows, the event that caused the overflow is taken as a sample [19]. The relational profiling architecture [15] and Zilles and Sohi's co-processor for profiling [23] use sampling to reduce the number of events that their special hardware must analyze, so that it can better keep up with the main processor.

Software sampling typically uses events that happen infrequently but regularly. The Jikes RVM takes samples on thread switches [1]. In ephemeral profiling, profiling is enabled from a timer interrupt and disappears on counter overflow [21]. Harris collects information relevant for prefetching by sampling objects whose allocation leads to an overflow of the local allocation buffer [14]. Whaley uses the operating system *sleep* function to disable the profiler thread between taking samples of the call-stack [22]. Arnold and Sweeney suggest sampling based on global or individual method counts as an alternative for this [5]. The Arnold-Ryder framework samples using counters in checks at all procedure entries and loop back-edges [4]. We also sample using counters in checks, but place checks in fewer places than Arnold and Ryder to reduce the overhead of sampling.

6.2 Causality Profilers

To reduce overhead, profiling typically abstracts the information available about the executing program. A common abstraction is to give up flow sensitivity: the profile abstracts from the order in which the instructions of the program were executed. A related abstraction is to give up context sensitivity: the profile of a routine abstracts from the calling context in which it was executed. When

% overlap

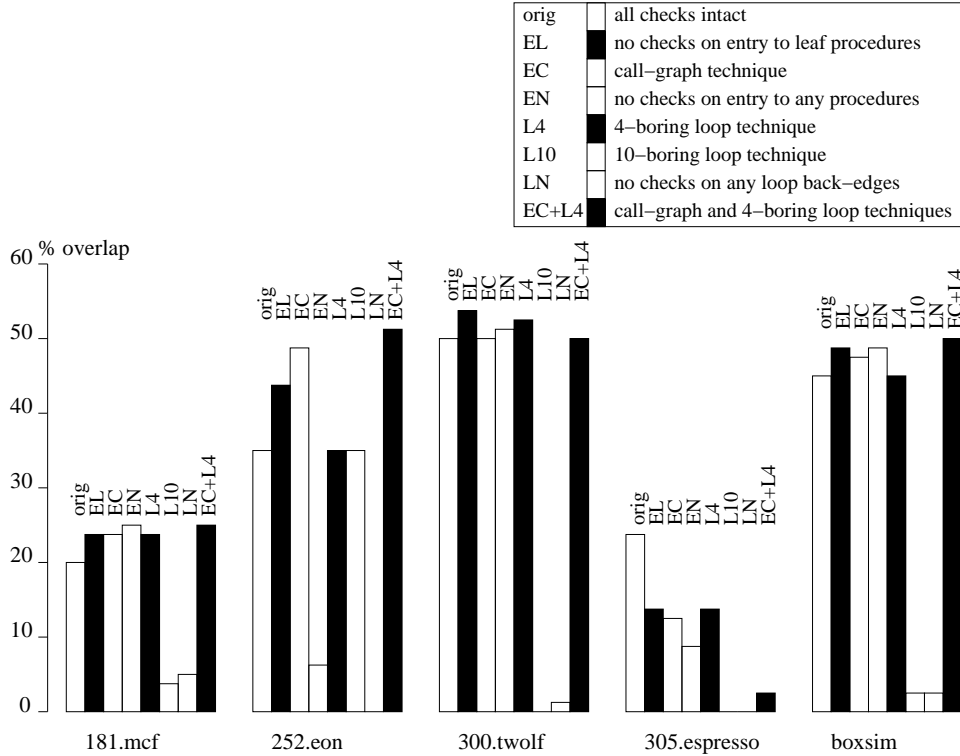|	
orig	all checks intact
EL	no checks on entry to leaf procedures
EC	call–graph technique
EN	no checks on entry to any procedures
L4	4–boring loop technique
L10	10–boring loop technique
LN	no checks on any loop back–edges
EC+L4	call–graph and 4–boring loop techniques

Figure 8: Overlap of hot data streams for various overhead reduction techniques.

both abstractions are used together, profiling events are viewed in isolation, and optimizations that exploit causality between events lack crucial information. Hence, various recent papers describe alternative abstractions that do not give up causality in the profile.

Ball and Larus describe an efficient technique for finding how frequently each acyclic path through the control flow graphs of individual procedures is executed [7]. This provides intraprocedural optimizers with important control flow information beyond edge profiles.

A number of profilers inspect the state at the moment when the profile is taken to infer information about the recent past to gain some limited causality information. DCPI and ProfileMe inspect the state of the main processor to find instructions that are simultaneously in flight and to correlate instructions with events like misprediction stalls [3, 12]. They describe the guess-work done in software to recover missed information. ProfileMe also uses the global branch history available in the processor to capture temporal relationships [12]. Ammons *et al.* describe the calling context tree, a representation that captures context-sensitive information, and describe how to combine it with flow sensitive information for intraprocedural acyclic paths [2]. Whaley [22] and Arnold and Sweeney [5] show how to find approximations to the calling context tree with low overhead by sampling call stacks.

To summarize, the two main software techniques for temporal profiling are profiling intraprocedural acyclic paths [7] and finding the calling context tree [2]. Intraprocedural acyclic paths capture flow-sensitive information, but miss temporal relationships between events that are separated by a procedure entry or a loop back-edge. Calling context trees capture context-sensitive information, but miss information across non-trivial sequences of calls and returns.

Larus describes whole program paths that contain more complete flow- and context-sensitive control relationships [17]. Larus uses the Sequitur algorithm [18] to compress the profile information, and shows how to find hot subpaths on the compressed representation. Chilimbi uses similar techniques to capture flow- and context-sensitive data relationships [8]. These techniques deal with the limitations of intraprocedural acyclic paths and calling context trees for temporal profiling, but at the cost of a much higher overhead.

Our bursty tracing profiler is intended for use in combination with Larus's and Chilimbi's profile representation and analysis techniques. We reduce their overhead by sampling, thus making them practical for dynamic optimization. The case study in Section 5.2 is an example for combining our bursty tracing technique with Chilimbi's representations and abstractions of data reference profiles.

7 Conclusions

This paper describes our enhancements to the Arnold-Ryder framework that make it suitable for low-overhead temporal profiling. We extended it to sample longer bursts that may span procedure boundaries. This enables us to collect temporal profiling information that is interprocedural and context-sensitive. We implemented it for x86 binaries. This enables us to use it for a different class of applications. We reduced its basic overhead. This made it practical for dynamic optimizations on x86 binaries to use temporal profiling information. We are working on a dynamic optimizer that improves data cache behavior by detecting and prefetching hot data streams found with our profiler.

Acknowledgements

References

[1] Bowen Alpern et al. The Jalapeno virtual machine. *IBM Systems Journal*, 2000.

[2] Glenn Ammons, Thomas Ball, and James Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Programming Languages Design and Implementation (PLDI)*, 1997.

[3] Jennifer Anderson et al. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)*, 1997.

[4] Matthew Arnold and Barbara Ryder. A framework for reducing the cost of instrumented code. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[5] Matthew Arnold and Peter Sweeney. Approximating the calling context tree via sampling. Technical Report RC-21789, IBM Research, 2000.

[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[7] Thomas Ball and James Larus. Efficient path profiling. In *International Symposium on Microarchitecture (MICRO)*, 1996.

[8] Trishul Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[9] Trishul Chilimbi. On the stability of temporal data reference profiles. In *Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[10] Trishul Chilimbi, Bob Davidson, and James Larus. Cache-conscious structure definition. In *Programming Languages Design and Implementation (PLDI)*, 1999.

[11] Trishul Chilimbi and James Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management (ISMM)*, 1998.

[12] Jeffrey Dean *et al.* Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture (MICRO)*, 1997.

[13] Dean Deaver, Rick Gorton, and Norm Rubin. Wiggins/Redstone: An on-line program specializer. In *Hot Chips*, 1999.

[14] Timothy Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM)*, 2000.

[15] Timothy Heil and James Smith. Relational profiling: Enabling thread-level parallelism in virtual machines. In *International Symposium on Microarchitecture (MICRO)*, 2000.

[16] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *International Symposium on Computer Architecture (ISCA)*, 1997.

[17] James Larus. Whole program pahts. In *Programming Languages Design and Implementation (PLDI)*, 1999.

[18] Craig Nevill-Manning and I.H. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *Journal of Artificial Intelligence Research*, (7):67–82, 1997.

[19] S. Subramanya Sastry, Ratislav Bodík, and James Smith. Rapid profiling via stratified sampling. In *International Symposium on Computer Architecture (ISCA)*, 2001.

[20] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[21] Omri Traub, Stuart Schechter, and Michael Smith. Ephemeral instrumentation for lightweight program profiling. Technical report, Harvard University, 2000.

[22] John Whaley. A portable sampling-based profiler for Java virtual machines. In *Java Grande*, 2000.

[23] Craig Zilles and Gurinder Sohi. A programmable co-processor for profiling. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2001.