

# The Performance Potential of Trace-based Dynamic Optimization

Brian Fahs Aqeel Mahesri Francesco Spadini  
Sanjay J. Patel Steven S. Lumetta

Center for Reliable and High Performance Computing  
University of Illinois at Urbana-Champaign

## Abstract

*Dynamic optimization can apply powerful optimizations to hot execution paths that span traditional boundaries such as branches and calls, including calls to dynamic libraries. Because of this opportunity, it has gained academic and industrial attention.*

*In this paper, we examine dynamic optimization in an overhead-free environment to ascertain its optimization potential. Specifically, we explore an ideal implementation with zero overhead and ask: what is the performance potential of trace-based dynamic optimization? Using a methodology consisting of a trace selector, optimizer, and trace-driven simulation framework, we provide insights on this potential.*

*We begin by demonstrating the potential of dynamically-applied classical optimizations as a function of trace length. Dynamic optimizers identify hot execution paths better than static mechanisms and, as a side-effect, are able to create longer optimization regions. In this analysis, we quantify contributing performance factors and identify optimization hindrances and their cost. We perform cross-ISA analysis of SPARC and x86, demonstrate significant differences between the two, and explain the basis for the differences. We also examine performance, trace cache size, and instruction stream coverage effects for ideal and real-world trace selectors and estimate the remaining optimization potential available.*

## 1 Introduction

Dynamic optimizers apply code transformations to an executing application. Applying optimizations dynamically provides several benefits over static optimization, particularly for control-intensive programs. For example, dynamic optimizations span branches and calls, including calls to dynamically-linked code, which typically hinder static optimization. In many cases, a dynamic optimizer is able to identify and optimize longer hot execution paths than a static mechanism. Also, because it naturally and transparently profiles during execution, a dynamic optimizer can adapt to changing program behavior.

Partly because of the potential for substantial performance improvements, there has been significant recent activity in both software- and hardware-centric dynamic optimization [2, 4, 6, 8, 13, 14, 16, 21, 23, 25, 30]. While each of these schemes is different in its approach, most schemes share a common structure that involves four processes: (1) region selection, (2) optimization, (3) optimized region storage, and (4) dispatch of optimized,

cached regions during execution.

Optimization regions are connected subgraphs of a program’s flow graph, such as traces [15], trees [22], or superblocks [20]. Certain optimizers, particularly those coupled to dynamic translators and just-in-time compilation systems, form regions based on higher-level program constructs such as methods or procedures.

In this paper, we ask the question: what is the performance potential of an ideal trace-based dynamic optimizer that incurs no overhead in selection, optimization, storage, or dispatch of traces? Dynamic optimizers derive performance benefit from code optimizations and from realignment of non-contiguous code. By eliminating all overheads and increasing optimization span to longer traces, we are able to bound the potential of trace-based dynamic optimization. Despite significant dynamic optimization research and development, little is understood of its performance potential. This lack of knowledge is partly due to implementation complexity. Design choices made during implementation can place artificial limits, which make general potential estimation difficult.

The results presented in this paper are divided into two studies. In the first study, we examine speedup potential as a function of trace length. Specifically, we estimate the incremental performance improvement due to optimizing traces consisting of longer paths, thereby mimicking an optimizer that is better at forecasting program control. Our results demonstrate that atomic traces, i.e., ones with no early side exits, permit deeper optimization than non-atomic traces, but at best provide a speedup of 2x even under very ideal situations. We also observe that memory aliasing remains a significant impediment to performance, and infer that optimizers able to determine or correctly guess dependence relationships will produce higher performance. We perform cross-ISA examination of optimizer potential, on SPARC and x86, and observe very different performance potential based on ISA. We examine the fundamental reasons for such differences introduced at the ISA level.

Our second study examines the impact of trace selection on performance, coverage, and trace cache size. In this study, we examine optimization potential and trace selection by implementing one oracular and two realistic trace selectors from recent literature that heuristically select traces based on expected coverage. Through these experiments, we notice that a marginal increase in performance and coverage requires an exponential increase in the number of instructions to be optimized, indicating that overhead costs might ultimately limit performance potential. We observe that realistic trace selectors provide similar performance, coverage, and trace cache sizes to the ideal oracular scheme. Finally, our experimental data indicates that partial selection of traces during fetch is an important ingredient for performance.

Section 2 provides a clarification of terminology used throughout this paper. Section 3 describes our experimental infrastructure. Section 4 reports the results of our experiments and analysis on speedup potential as a function of trace length. Section 5 contains our trace selection experiments. Section 6 discusses the related work. Conclusions are provided in Section 7.

## 2 Conventions and Definitions

For the reader’s convenience, we define terms and conventions used in this paper.

**dynamic optimizer:** A system that optimizes programs while they run. We only examine trace-based dynamic optimizers in which a trace selector and a trace optimizer are the primary components.

**low-level optimizations:** Binary-based optimizations lacking high-level information, such as type information. We only examine these optimizations.

**trace:** A sequence of basic blocks (possibly with repeated blocks) containing no internal control flow and a single entry point.

**non-atomic trace:** A trace with multiple exit points.

**atomic trace:** A trace with a single exit point. Atomicity can be provided in various ways, such as hardware checkpointing [26] or recovery code [15, 20].

**trace selector:** An algorithm that selects traces to be optimized from the dynamic instruction stream.

**coverage:** The percentage of the original dynamic instruction stream that executes from dynamically selected traces.

### 3 Experimental Infrastructure

We remove dynamic optimization overheads by including an optimizer in our performance simulator. This framework allows us to pass arbitrary optimized (or unoptimized) instruction sequences to our processor timing model.

#### 3.1 Performance Simulator

Our simulator infrastructure consists of SPARC and x86 instruction trace files, a translator that decodes instructions into our micro-operation ISA, and a parameterized timing simulator.

We used eight SPECint 2000 instruction traces for SPARC and fourteen for x86; seven of the fourteen for x86 are SPECint 2000 and seven are Winstone desktop application traces. Table 1 contains a summary of the workloads. We used the Shade [7] instruction tracing program to gather our SPARC instruction traces from heavily optimized binaries. The x86 instruction traces were graciously provided by AMD and consist of execution “hot spots” generated on a Windows NT hardware tracing platform from heavily optimized binaries. For benchmarks common to SPARC and x86, we were not able to ensure that the instruction traces for both ISAs were gathered from the same point in execution, however, it is possible that some overlap does exist.

For most of the x86 benchmarks that we excluded, we do not have their instruction trace files and have no way to generate them. There were four benchmarks, three SPECint and one Winstone, that we did have trace files for, but due to problems with our infrastructure were unable to simulate their performance. For SPARC, we were unable to gather instruction traces for the other SPECint benchmarks because of problems with the Shade

Category	Applications	Avg. Orig. Insts.	Avg. Uops
SPARC SPECint	bzip2, crafty, eon, gap, mcf, parser, twolf, vpr	49M	56M
x86 SPECint	bzip2, gzip, crafty, eon, parser, twolf, vortex	51M	61M
x86 Winstone	Access, Dreamweaver, Excel, Lotusnotes, Photoshop, Powerpoint, Soundforge	101M	132M

**Table 1. Experimental Workload**

Fetch/Decode/Issue/Retire	8 instructions per cycle
BrPred	18-bit gshare, 1K-entry BTB
Pipeline	Dynamically scheduled, 15 cycles (min) for BR res
Inst Window	512 instructions
ExeUnits	8 IALUs, 2 IMULs, 3 FLTs, 4 Loads, 4 Stores
L1 I Cache	64K, 2-way assoc., 32B line size, 2 ports, 1 cycle
L1 D Cache	64K, 4-way assoc., 32B line size, 2 ports, 3 cycles
L2 Unified Cache	1M, 2-way assoc., 128B line size, 10 cycles
Memory	200 cycles

**Table 2. Simulated Machine Configuration.**

program. Because we do not have the source for Shade, we were unable to fix the problems we encountered. We did not try to gather SPECfp traces.

To use a single optimizer, we translate instructions for SPARC and x86 into a micro-operation ISA. Each micro-instruction performs simple, RISC-like operations with support for 64-bit immediates. Our translator converts each SPARC instruction into an average 1.14 micro-operations and each x86 instruction into 1.25 micro-operations. The SPARC expansion results from differences between SPARC and our ISA<sup>1</sup>. The x86 expansion comes from the complex nature of the x86 instruction set.

Using our own ISA allows the number of architectural registers to be parameterized. This feature enables the optimizer to use a much larger register set than the original architectural registers. The optimizer need only ensure architectural correctness for memory and registers of the original ISA. We eliminate register pressure in our experiments by using 2,048 architectural registers, which we empirically observed to act equivalently to an infinite register set for all data we report. An unlimited register set avoids any artificial performance limitation imposed by architectural register file size or optimizer register allocation algorithm, and it results in an optimistic upper bound on performance, which is the intent of this study.

Our timing model, summarized in Table 2, simulates an 8-wide, out-of-order, superscalar processor with a 15-cycle minimum branch resolution latency, and is arguably representative of current or next-generation processors.

<sup>1</sup>For example, `register+register store` instructions in SPARC are expanded into two micro-operations.

## 3.2 Optimizer

The heart of our infrastructure is our optimizer. As stated previously, we compare non-atomic and atomic optimization. In this subsection, we compare the two approaches and describe the optimizations performed by our optimizer.

### 3.2.1 Non-atomic vs. Atomic Optimization

Non-atomic and atomic traces are not optimized identically, but both allow optimization of later blocks in a trace to use information derived from earlier blocks, such as known constants. Unlike atomic traces, non-atomic traces do not allow information from later blocks, such as overwritten architectural registers, to be used to optimize earlier blocks because side-exits can be taken. Non-atomic optimization is not the same as trace [15] or superblock [20] scheduling (aggressive forms of non-atomic optimization), which act more like atomic optimization through the use of recovery code. It should be noted that atomic traces can also be supported directly in hardware, as is the case with frames in [26].

### 3.2.2 Low-level Optimizations

Our optimizer performs several low-level compiler optimizations. While many optimizations are possible, these optimizations were chosen because dynamic optimizers use them [2, 4, 12, 13].

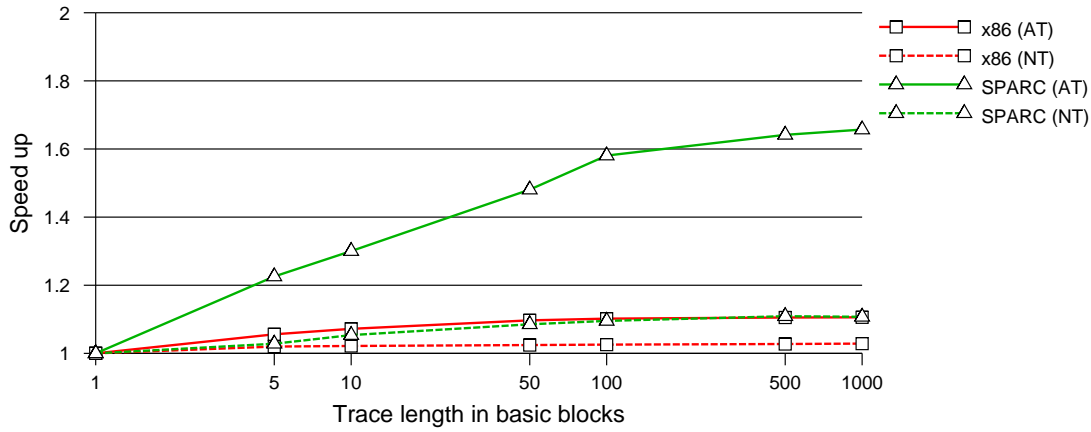
**NOP removal** eliminates instructions that produce no results or side-effects. This optimization is performed for all experiments, including baseline measurements.

**Dead code removal** eliminates instructions, including stores, whose results are overwritten without being used within a trace. Often, even heavily optimized programs contain dead code, but only on specific control paths. Store instruction removal, although too optimistic for real systems, is appropriate given the nature of our work. This optimization also removes unconditional branch instructions.

**Constant propagation** breaks register dependences, increases ILP and decreases computation tree height by forwarding constants and value ranges to later instructions. For atomic traces, this optimization also backward propagates constants derived from later branch directions.

**Reassociation** combines associative operations to reduce computation tree height and increase ILP. It also aids function inlining by collapsing stack pointer manipulation, thereby enabling store forwarding and dead code removal to eliminate pushes and pops on the call stack. This optimization also subsumes copy propagation.

**Common subexpression removal** eliminates redundant computations, such as redundant load instructions. Our implementation encompasses value numbering as well.



**Figure 1. Speed up as a function of trace length relative to non-dynamically optimized execution.**

**Store forwarding** propagates a store input to a dependent load’s dependents when no intervening conflicting memory operations exist. `register±offset` expressions, first simplified by the aforementioned optimizations, are used to identify dependence, and conflicts are assumed when no relationship exists. This optimization is referred to as load-store telescoping in [12].

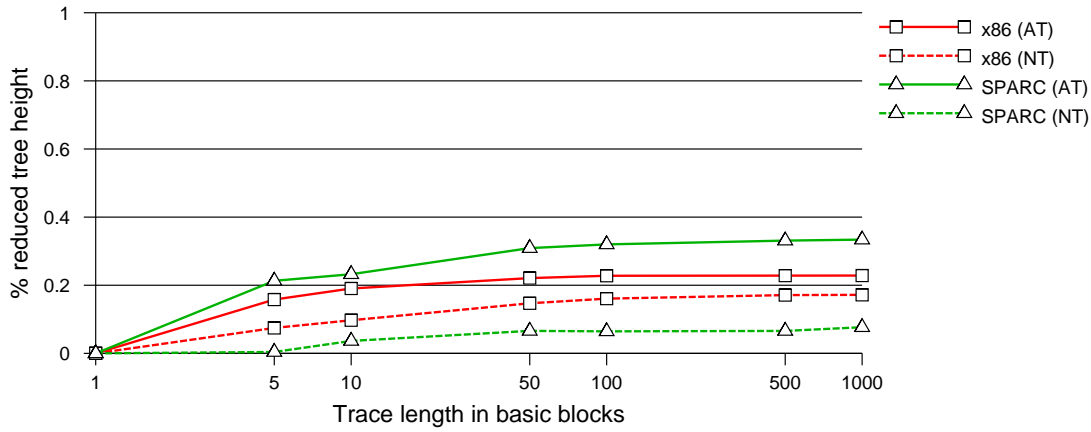
**Strength Reduction** converts degenerate long latency operations into shorter, simpler operation sequences. Primarily, it converts integer multiplications by powers of two into left shifts.

**List Scheduling** schedules optimized traces for the simulated processor’s constraints. This optimization maximizes pipeline throughput.

#### 4 Performance vs. Trace Length

In this section, we assess the impact of optimization without regard to trace selection by evaluating optimization effectiveness as a function of trace length, where trace length is varied from 1–1000 basic blocks. While this evaluation approach is good for measuring raw optimization performance (i.e., it provides 100% coverage), it yields almost no trace reuse. Without reuse, performance suffers due to numerous unique trace instruction addresses that miss the instruction cache. Consequently, this study warrants a perfect fetch model where the instruction cache always hits and the processor fetches the full width of instructions unless the instruction window is full. Because there are no instruction caching effects, we do not emphasize performance numbers in this section, but rather provide trends which demonstrate optimization impact. Figure 1 plots average improvement for SPARC and x86 as a function of trace length. NT denotes non-atomic trace optimization, AT denotes atomic trace optimization.

Although not demonstrated in the figure, our translator automatically optimizes single basic blocks during translation. We did not include the translator optimization effects for any of our performance graphs because we did not want to artificially inflate results with inefficiencies introduced by translation. However, we would



**Figure 2. Tree height reduction for SPARC and x86 benchmarks.**

like to note that the translator provided a speed up of 1.18 for x86 and 1.11 for SPARC<sup>2</sup>. Intuitively, for traces of length one, SPARC should have little improvement because translation overhead is small, i.e., 1.14 micro-instructions per SPARC instruction. The most significant contributing factor is that our micro-operations provide 64-bit immediates, which allows multiple instructions that originally produced a single constant to be converted to one instruction. This benefit is not as pronounced for x86 because a single x86 instruction can load 32-bit (full-sized) immediates. The single block optimization benefits for x86 originate from simplifying inefficiencies in CISC-to-RISC translations, e.g., combining ESP register (the x86 stack pointer) updates from multiple push instructions into a single ESP register update.

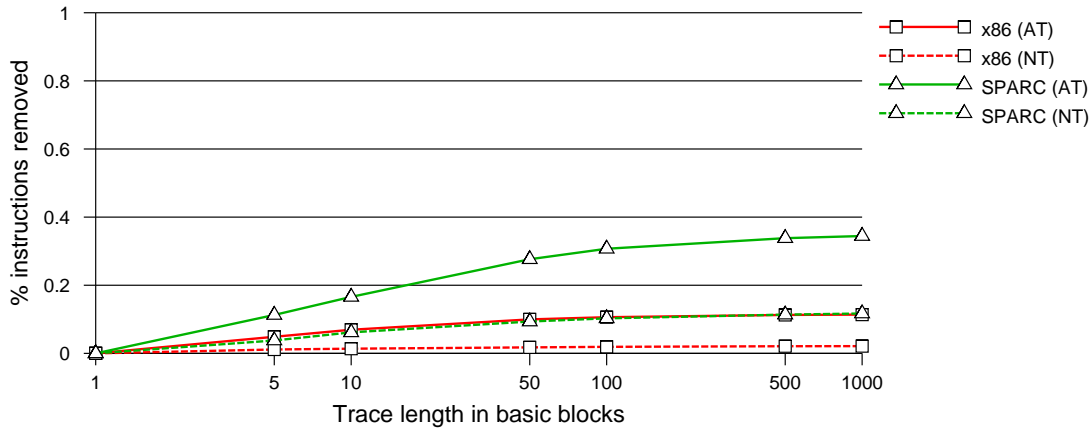
In the following subsections, we divide the improvements in Figure 1 into contributing factors and analyze the trends and behavior.

#### 4.1 Performance Factors

Optimization benefit primarily comprises two components: computation tree height reduction and instruction count reduction. Optimizations like reassociation reduce computation tree height. Dead code removal reduces instruction count, although other optimizations increase its effectiveness.

To understand how optimization affects computation tree height, we measured optimization impact on dynamic dataflow graph tree height. For this measurement, all caches are perfect, hardware resources are unlimited, and only true data dependences are obeyed. Functional unit latencies are used for dataflow graph edge weights. A similar study with similar optimizations was performed in [12], but the results are presented in a different way and serve a different purpose. Our oracle parallelism results (not shown) are comparable to theirs even though we used a newer set of benchmarks and different ISAs.

<sup>2</sup>Without translator optimization, SPARC and x86 measurements would be scaled versions of those shown here.



**Figure 3. Instruction count reduction for SPARC and x86 benchmarks.**

Figure 2 shows average computation tree height reduction for x86 and SPARC as a function of trace length. Because our translator optimizes single basic blocks to remove translation inefficiencies, 28% (not shown in figure) of the original tree height was removed from the baseline. An additional 23% of the remaining tree height was removed from atomic traces as trace length extends to 1000 basic blocks. The combination of these two provides a total 41% reduction of original tree height (1.69 speed up on an infinitely wide machine).

Unlike x86, the translator only removed 1% of the original tree height during SPARC translation. As trace length extends to 1000 basic blocks, 33% of the remaining tree height can be removed resulting in a total reduction of 34% for atomic traces.

Figure 3 plots average percentage of micro-instructions removed as a function of trace length. For x86, the translator removed 9% of the original instructions; for SPARC, it removed 13%. Atomic x86 optimization and non-atomic SPARC optimization coincidentally produce an almost identical curve. Atomic and non-atomic SPARC reduce more instructions than atomic and non-atomic x86, respectively.

For both ISAs, atomic optimization provides better speed ups and reduces both computation tree height and instruction count more than does non-atomic optimization.

Earlier, we showed that speed ups on our machine model exhibit diminishing returns with increased trace length. We have now also shown that tree height reduction and instruction count reduction have similar trends. These trends indicate that the impact of optimization is actually tapering off, and that the trends in Figure 1 are not just a result of bottlenecks within our simulated machine.

Table 3 provides average peak speed ups for SPARC and x86 for three machine configurations. The infinitely wide and single wide machine speed ups assume perfect memory and remove all real-life bottlenecks; they are derived from the tree height and instruction count reduction measurements in Figures 2 and 3. Because our machine model takes advantage of both tree height and instruction count reduction, its performance can actually be higher than that produced by either the infinitely wide machine or the single wide machine. It can also be lower

ISA	Speed up		
	Our machine model (perfect fetch)	Infinitely wide machine (tree height reduction)	Single width machine (instruction count reduction)
SPARC	1.66	1.50	1.53
x86	1.11	1.29	1.13

**Table 3. Speed ups for ideal machines with atomic traces.**

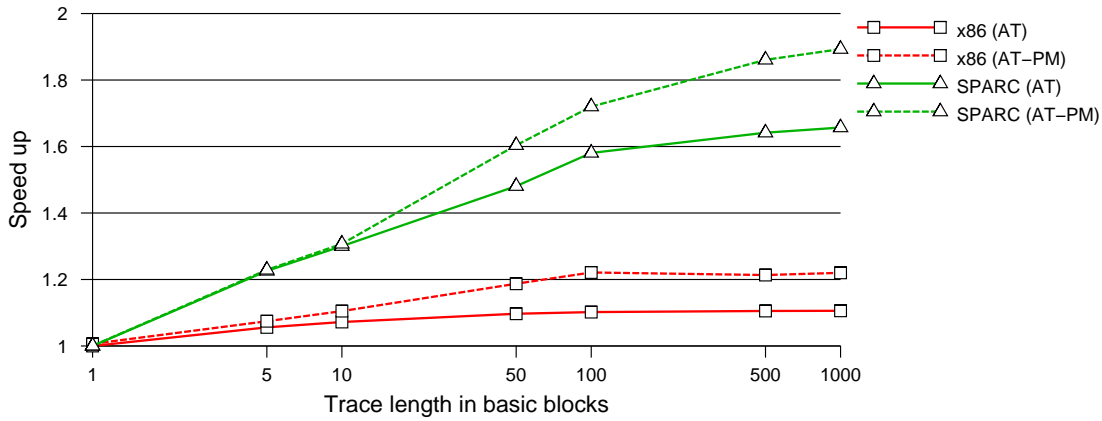
than both because of bottlenecks in our simulated machine. While one may contrive situations or simulators that produce larger improvements, speed up due to optimizations alone does not generally exceed 2x simply because the effects of optimization, i.e., tree height and instruction count reduction, do not support larger speed ups. Of course, this limitation is predicated on the quality of the baseline and on the optimizations performed. Our baseline programs are compiled with static optimization; unoptimized binaries could produce different improvements. Our optimizations were selected based on what is being used in dynamic optimization. Other optimizations could potentially alter the outlooks that we provide here. We expect achievable speed ups from optimization to be even lower than those presented here because we are performing an idealized study; overheads, interrupts, and exception handling will potentially dilute the performance impact. Why are speed ups not higher? The low-level optimizations lack high-level semantic information and, therefore, have limited capabilities. Even for high-level optimization, a certain amount of instructions and computation tree height are required to execute an application correctly. In Section 4.2.2, we look at the impact of ambiguous memory accesses on optimization potential as a possible limitation, but we do not see conclusion-altering results from that experiment.

## 4.2 Analysis

In Figure 1, atomic trace optimization demonstrates significant improvements for SPARC and moderate improvements for x86. Although lower than atomic, non-atomic optimization for SPARC and x86 provides similar, moderate improvements. Overall, SPARC and x86 have marginal improvements for exponential trace length increases, ultimately leveling off at trace lengths of 100 basic blocks for SPARC and 5 or 10 basic blocks for x86. In the following subsections, we analyze these characteristics through experimentation and discussion. Specifically, we analyze reasoning for divergence of non-atomic and atomic optimization, limiting factors to further improvement, and reasoning for SPARC and x86 differences.

### 4.2.1 Atomic vs. Non-Atomic Potential

We have shown that large differences exist between non-atomic and atomic optimization, but have not explained the reasons for these differences. The dissimilarity results from non-atomic traces requiring live-out registers of each basic block to be preserved. Requiring preservation of each block’s live-outs necessarily limits register lifetimes. This limitation constrains optimizations that reduce tree height and remove instructions because they increase register lifetimes during optimization. Additionally, dead code removal can not remove dead instructions



**Figure 4. Impact of perfect memory disambiguation.**

that are live-out for a block. Trace [15] and superblock [20] scheduling reduce live-out restrictions through side-exit recovery code, and, therefore, can approach atomic optimization performance.

#### 4.2.2 Impact of memory aliasing

For most binary-based optimizers, the ambiguity of memory operations presents barriers to optimization. In this section, we investigate memory disambiguation’s impact on performance by allowing perfect disambiguation, i.e., we remove all optimization limitations due to memory accesses. Because our simulation workloads are instruction traces, we are able to allow the optimizer to peek at load and store addresses during optimization (i.e., prior to execution). Figure 4 presents performance for atomic trace optimization with perfect memory disambiguation as a function of trace length. AT-PM is atomic trace optimization with perfect memory disambiguation. The atomic trace optimization (AT) measurements from Figure 1 are included for comparison.

SPARC and x86 improvements are significantly better for AT-PM. Not only is performance better, but the trends begin to show diminishing returns at much higher trace lengths. For SPARC, atomic trace optimization began to saturate at trace lengths of 100, but perfect memory disambiguation extends this limit to 500 or 1000. For x86, atomic trace optimization saturates at trace lengths of 5 or 10, but perfect memory disambiguation increases it to 100.

Why does the lack of memory disambiguation hinder performance so much? The ambiguous nature of memory operations often form barriers to simple optimizations. As trace size increases, optimizations can process longer chains of computation. But once an entire computation chain, starting and ending at memory operations, is captured within a trace, there is little benefit to that chain in enlarging the trace. However, by disambiguating memory operations, chains of computation can be connected together, which allows for optimizations to span computation chains making them much more effective.

For tree height reduction and instruction count reduction we also witnessed diminishing returns for increased

ISA	Speed up					
	Our machine model (perfect fetch)		Infinitely wide machine (tree height reduction)		Single width machine (instruction count reduction)	
	AT	AT-PM	AT	AT-PM	AT	AT-PM
SPARC	1.66	1.89	1.50	2.26	1.53	1.82
x86	1.11	1.22	1.29	1.87	1.13	1.44

**Table 4. Speed ups for ideal machines with atomic traces and perfect memory disambiguation.**

trace length for both ISAs. Table 4 shows the speed ups with atomic optimization and perfect memory disambiguation that are possible on our machine model, an infinitely wide machine, and a single wide machine. It is important to notice that, even with perfect memory disambiguation, speedup hovers around 2x (slightly above for SPARC on an infinitely wide machine). For x86, performance improvement doubles, tree height reduction is 2x, and instruction count reduction is 2.66x larger with perfect memory disambiguation. SPARC has only a 35% improvement with our machine model when perfect memory disambiguation is used. SPARC’s tree height reduction is 1.67x larger and its instruction count reduction is 1.89x larger when compared to atomic optimization without perfect memory disambiguation.

### 4.2.3 SPARC vs. x86

Throughout Section 4, we demonstrated a clear disparity between SPARC and x86 performance, i.e., speed ups and optimization effectiveness for SPARC seem far superior to x86. Because we use a common dynamic optimizer and micro-instruction set, we expect absolute IPC for SPARC and x86 to be roughly equivalent after a sufficiently large trace length. However, the benchmarks and sampling points are different and, therefore, absolute performance can differ. Using gcc<sup>3</sup> version 3.3 with optimization level -O3, we removed the compiler and sampling point differences by compiling twolf for SPARC and x86 and creating two semantically identical execution traces<sup>4</sup> of the subroutine that used the largest fraction of execution. When we measured the performance of these two traces, tree height and instruction count were indeed very similar for both ISAs, particularly after perfect memory disambiguation because disambiguation is important for x86 due to the limited architectural register set. However, SPARC had a relative performance improvement significantly larger than x86. But why?

Some restrictions of the SPARC ISA, e.g., limited addressing modes and small immediates, result in a significantly larger baseline computation tree height and instruction count. While much of these inefficiencies are absorbed in single block optimizations performed by the translator, the larger architectural register set of SPARC allows the compiler to more easily extend register definition and use across basic blocks, which can cause translation inefficiencies to remain. The limited architectural register set of x86 primarily confines these inefficiencies to a single block. Our micro-instruction set was originally designed for x86. It supports the complex addressing modes of x86 and instructions like `lea`, which adds two registers and an immediate. While we did incorporate

<sup>3</sup>We performed the same analysis using Sun’s compiler for SPARC and found consistent results.

<sup>4</sup>To generate the x86 trace, we used the Bochs x86 functional simulator instead of the AMD traces.

extensions for SPARC, supporting the complex x86 instructions provides some inherent instruction combining opportunities for SPARC that exist only because the SPARC ISA is a more restricted instruction set. Additionally, unlike x86, SPARC is able to perform aggressive code motion at compile time due to its large architectural register set, which potentially produces partially dead code.

We acknowledge that the SPARC performance numbers are potentially higher than possible. What does that mean for our evaluations? The speed ups, tree height reduction, and instruction count reduction estimates, which have been demonstrated to be much larger than x86, are likely too high and may actually be much closer to the x86 estimates. If SPARC were first translated to a micro-instruction set with relaxed memory addressing modes and larger immediates, the measurements we present would be more accurate.

It is interesting to note that many architects think of translating x86 (CISC) instructions to a micro-instruction set at least partially because of opportunities to remove inefficiencies. Our experience indicates that there is probably equal opportunity for improvement in the SPARC ISA (RISC).

## 5 Trace Selection

An ideal trace selector provides optimal performance for a set of optimization resources. Unfortunately, ideal trace selection is probably not computable, particularly once finite resources and dataflow interactions between specific basic blocks are factored in. As an alternative to optimal selection, we explore heuristic approaches to trace selection and evaluate their performance. In the following subsections, we evaluate the impact of trace selection on performance, coverage, and the number of instructions that must be optimized.

### 5.1 Trace Selection Algorithms

For our evaluations, we use three trace selectors. One algorithm is oracular, meaning selection occurs with advanced knowledge of the entire dynamic instruction stream; two are realistic schemes used in prior dynamic optimization work. Three other algorithms were evaluated,<sup>5</sup> but excluded because they provided poor performance, coverage, trace cache size, or a combination of the three. Below we describe the three algorithms evaluated in this section:

**Min-Traces** attempts to minimize the number of traces. Given the complete execution sequence and a target length  $k$  in blocks, traces (of any length up to  $k$ ) with highest coverage are selected and all its instances are removed from the execution sequence. The process then repeats, selecting from the remaining execution sequence.

**Threshold** is an implementation of rePLay’s frame construction algorithm [13], which bases trace construction on branch bias. A branch which proceeds in the same direction 32 consecutive times is considered biased.

---

<sup>5</sup>We also evaluated a dictionary-based compression scheme, a scheme that allowed any previously seen trace to be optimized, and another oracular scheme that chooses the best traces that are exactly  $k$  blocks in length.

Blocks are added to a trace repeatedly until the last branch is not biased or the maximum trace size is reached.

**Next-Executing Tail (NET)** is a low-overhead algorithm used in the Dynamo [10] and DynamoRIO [4] dynamic optimizers. When counters associated with backward-taken branch targets exceed a threshold, a trace is constructed in a single pass until the next backward-taken target is encountered. Although NET creates traces without regard to individual block frequency, it is statistically likely to select good traces [10]. In our evaluation, we unroll loops, as NET does not do so explicitly.

Unlike NET, min-traces and threshold do not assume partial trace matching. Partial matching allows subsets of a selected trace to be used, full trace matching requires that all or none of the trace be used. With partial matching, traces can exit early (single-entry, multiple exit). Partial matching can be important because it provides increased coverage using possibly smaller trace caches, which also potentially results in fewer instruction cache misses and better branch prediction.

Because of its potential importance, we also evaluated min-traces and threshold using partial matching. We evaluate partial matching in two ways. First, we partially match traces selected from the full matching algorithm. Secondly, we incorporate partial matching into selection; traces are chosen in the same manner, but selection criteria is not based on coverage of the entire trace, but coverage of the trace and all its subsets. We require a two block minimum for partial matching because a single block is better served from its original location. Incorporating single block traces would skew selection to the most popular single basic blocks, but these may not necessarily be good traces.

On the subject of optimization, partial matching implies non-atomic optimization, but using non-atomic optimization results in an overly pessimistic evaluation, as trace and superblock scheduling are non-atomic and have the potential to approach atomic optimization. Therefore, we optimize traces atomically for all evaluations in this section. Partially matched versions of a trace are atomically optimized separately from the entire trace, but given the same trace cache location (i.e., memory address) as the full trace. This setup, although unorthodox, allows all partially matched versions of a trace to be optimized in the best way possible and also retain and provide caching and branch prediction benefits with the other flavors of the trace. This exact organization is impossible in a real system, but trace scheduling and superblock scheduling could be used to provide very similar effects.

## 5.2 Experimental Setup

For each trace selector, we created many configurations for each benchmark by varying algorithm parameters. For each configuration, we created trace files, i.e., listings of traces selected by the configuration's algorithm. We evaluated performance for each trace file for 10 discrete coverage ranges: 0%-10%, 10%-20%, ..., 90%-100%. To control coverage, we repeatedly insert the trace with highest coverage into an initially empty trace cache until the desired coverage range is achieved. If a trace file does not produce coverage in one of the ranges, that range is simply discarded for that trace file. For example, many trace selectors cover over 90% of bzip2 with a single trace.

Algorithms	Parameter	Values	Partial/Full Match	Coverage
min-traces threshold	max. trace length	5, 10, 15, 20, 25, 50	partial & full	10 coverage ranges in 10% increments between 0% & 100%
	loop unrolling	5, 50		

**Table 5. Parameter variations for each selection algorithm**

Therefore, we did not evaluate those bzip2 trace files for coverage values lower than 90%. Algorithm parameters and coverage range information are summarized in Table 5.

For simulation, all traces are mapped to an in-memory trace cache located in an unused portion of the address space. Traces are sequenced into execution with zero-overhead, and next sequential prefetching is used to improve trace fetch bandwidth. In order to provide more realistic performance that accounts for the impact of trace caching and code realignment, we use the realistic fetch model described in Section 3.1. Specifically, we use a 64KB, 2-way set-associative instruction cache and an 18-bit gshare branch predictor with a 1K-entry BTB.

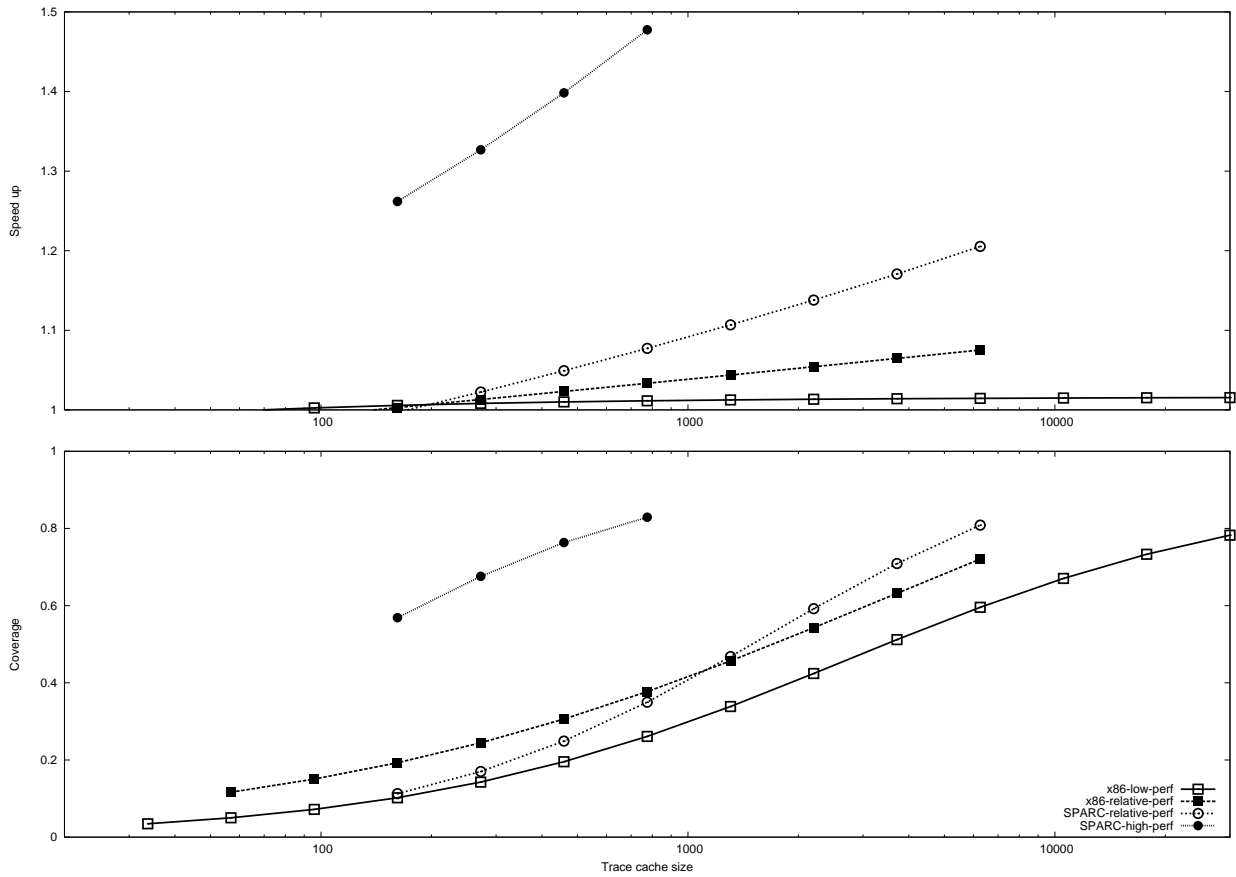
**NOTE:** We use the term trace cache size to denote the number of **original** instructions (not optimized instructions) that must be optimized. In a real system, where software or hardware trace caches require evicting traces, the number of instructions that must be optimized may increase due to re-optimization of previously evicted traces. The actual trace cache size in a real system may actually be much smaller due to the fact that not all traces may be needed all of the time. We do not attempt to measure the necessary dynamic trace cache size. Such an analysis is beyond the scope of this study.

### 5.3 Performance Evaluation

Simulating different trace selectors, parameters, and coverage ranges provided over 5,000 data points. Because of the overwhelming amount of data, through most of this section we only provide regression trends based on our analysis of all of the data. For all of the trends, we found that a Pearl-Reed curve [27], a member of the S-curve family, fit the overall trends well. We actually use a modified version of the standard curve to account for an exponential horizontal axis. In this section, we present trends that occur regardless of selection algorithm, compare trace selectors, and compare partial and full trace matching. We conclude this section with peak performance numbers and relate observed performance back to trends demonstrated earlier to indicate remaining optimization potential.

#### 5.3.1 Relationship Between Cache Size, Coverage, and Speed Up

Several prominent trends exist regardless of trace selection algorithm or parameter. Figure 5 plots four outstanding trends, two for SPARC and two for x86. The upper graph measures speed up and the lower measures dynamic instruction coverage. Trace cache size is measured on the common logarithmic horizontal axis. We examined performance vs. cache size and coverage vs. cache size for each benchmark, but for brevity, we consolidated



**Figure 5. Relationship between cache size, coverage, and speed up across all configurations.**

Group	Characteristics	ISA	Benchmarks
x86-low-perf	Large cache variation/coverage range and little performance improvement	x86	crafty, vortex access, lotusnotes
x86-mid-perf	Large cache variation/coverage range and performance improvement with cache size	x86	eon, parser, twolf dreamweaver, excel, powerpoint
SPARC-mid-perf	Large cache variation/coverage range and performance improvement with cache size	SPARC	crafty, gap, twolf, vpr
SPARC-high-perf	Small cache variation, high coverage high performance	SPARC	eon, mcf, parser

**Table 6. Breakdown of different performance/coverage vs. cache size groups.**

benchmarks into groups with similar trends. The groups are described in Table 6.

`x86-low-perf` produced a large range of trace cache sizes, from fewer than 100 instructions to greater than 10,000. Coverage varied from low to almost the entire instruction stream, but performance changed little. For these benchmarks, optimization and caching benefits provide small performance improvements even when the majority of instructions are optimized and originate from the trace cache.

`x86-mid-perf` and `SPARC-mid-perf` also produce large cache and coverage ranges. Increasing performance requires a corresponding increase in coverage and exponential increase in trace cache size.

`SPARC-high-perf` attain high coverage and high performance with small trace cache sizes.

Although 17 of our 22 benchmarks exhibited trends consistent with Figure 5, we also found outliers to the regression trends. `Bzip2` for `SPARC` and `x86` and `soundforge` exhibited high coverage for small trace caches (~100 instructions) but relatively no performance improvement. `Gzip` for `x86` had a relatively small cache size (~100-1000 instructions); performance and coverage increased significantly with trace cache size. `Photoshop`, which we know to have a small instruction footprint (at least for our instruction trace), performed very well for a small cache, but only when certain traces were selected.

Overall, most benchmarks demonstrate linear increases in coverage and performance for exponential trace cache size increases. This behavior implies that more powerful dynamic optimizers will have to be more efficient as overhead becomes increasingly difficult to recoup due to exponential increases in the number of instructions to be optimized.

### 5.3.2 Selection Algorithm Performance

Figure 6 compares trace selector performance. We only include results for `SPARC-mid-perf` to limit graph content. Most of the benchmarks had performance trends consistent with the graph, but for a few, `NET` required larger caches to provide identical levels of coverage and, in two cases, provided poor performance relative to the other trace selectors.

Although from the speed up graph it appears that `NET` outperforms both `min-traces` and `threshold` for certain trace cache sizes, it is actually not the case. Across their configurations, both `min-traces` and `threshold` had wide performance variations, which manifest themselves by lowering the overall trend. This is also the reason for the divergence of `min-traces` and `threshold` in the figure.

`NET` provides similar performance and coverage to the other trace selectors for significantly less overhead. These results are consistent with [10]. However, `NET` cannot scale to large coverages, and therefore is limited to lower speed ups and smaller trace caches. This shortcoming is directly caused by the fixed number of traces, which is controlled by the number of backward taken targets. These results reinforce the belief that `NET` is cost-effective, but indicate that more improvement is possible. Nevertheless, the ease with which `NET` creates traces provides hope for more cost-effective techniques to further improve performance and coverage without invasive profiling.

`Min-traces`, an oracular algorithm, provides very similar performance vs. trace cache size and coverage vs. trace cache size trends to `threshold` and `NET`. Assuming that `min-traces` is a good oracular trace selection algorithm, this

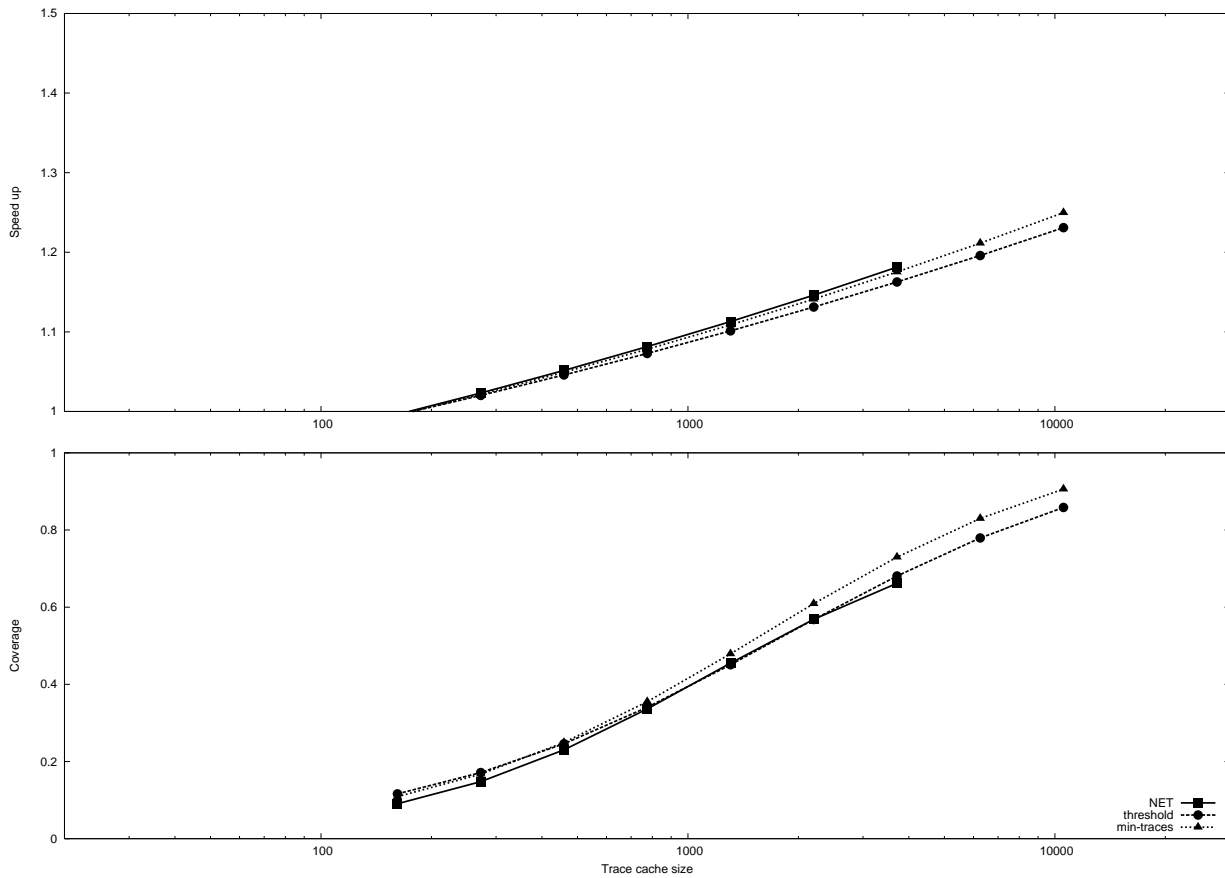


Figure 6. Heuristic selection scheme performance.

similarity indicates that algorithms currently in use in dynamic optimization are also good.

### 5.3.3 Partial vs. Full Matching Performance

Figure 7 compares partial and full trace matching for SPARC-mid-perf. With the exception of bzip2, all benchmarks exhibited similar trends.

Full trace matching provides less than half the performance improvement of partial matching. It is important to realize that trace scheduling or superblock scheduling would need to be used to approach the performance demonstrated here. In most systems, the instructions that are executed when a side-exit is taken would not be fetched along with the trace. This would result in lower performance than what we demonstrate due to extra instruction cache misses. It is possible, however, to prefetch side-exit recovery code with the trace and thus lower its negative impact.

Surprisingly, the coverage of partial and full matching do not vary significantly across the trace cache ranges. This seemingly non-intuitive behavior is actually a by-product of the way in which we select traces for partial matching for min-traces and threshold, which always increases trace length if coverage increases by doing so.

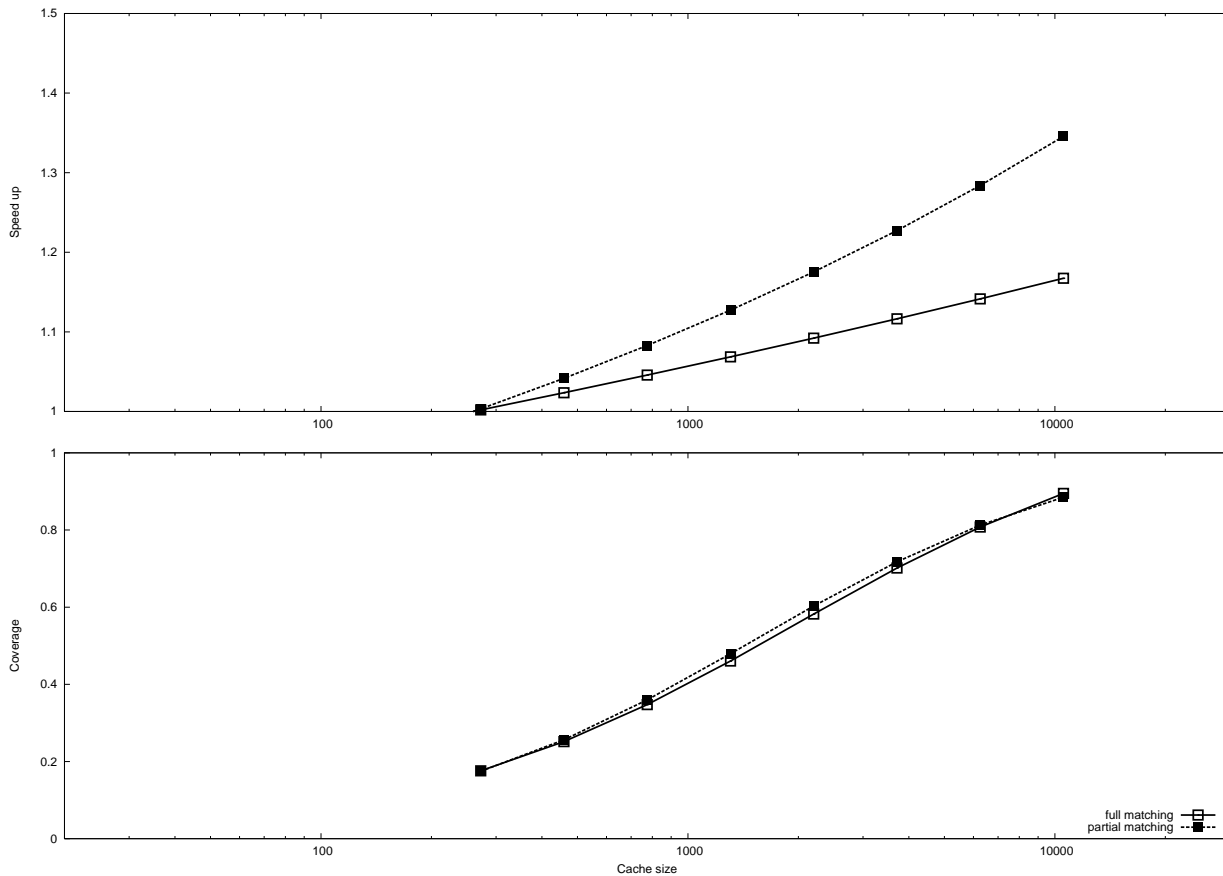


Figure 7. Partial matching performance.

Such a decision tends to increase the number of instructions that must be optimized.

### 5.3.4 Peak Performance

We have so far provided only performance trends. While trends are useful for observing performance, coverage, and trace cache size with respect to variables, like trace selector, they provide little indication of absolute performance. In this section, we provide our observed average peak performance numbers and link them to the performance vs. trace length trends we analyzed in Section 4. The peak performance number for a benchmark is the trace selection configuration (i.e., trace selector, parameters, and coverage) that provided the highest speed up.

Table 7 compares average speed up according to ISA. Peak realistic fetch is the average speed up for the peak configurations with our realistic fetch model. Peak perfect fetch is the average speed up of the peak configurations generated with our machine model with perfect fetch. Arbitrary blocking is the selection algorithm used in the first study where traces are constructed from every  $n$  basic blocks, where  $n$  is either 10 or 1000. As in the first study, arbitrary blocking is only simulated using a perfect fetch model. For the tree height and instruction count reduction tables, Peak represents the average measurement for the peak configuration (realistic and perfect fetch

Speed up comparisons				
ISA	Peak realistic fetch	Peak perfect fetch	Arbitrary blocking	
			@ 10	@ 1000
SPARC	1.62	1.29	1.30	1.66
x86	1.15	1.05	1.07	1.11

Tree height reduction comparison			
ISA	Peak	Arbitrary blocking	
		@ 10	@ 1000
SPARC	24%	23%	33%
x86	13%	19%	23%

Instruction count reduction comparison			
ISA	Peak	Arbitrary blocking	
		@ 10	@ 1000
SPARC	20%	17%	34%
x86	5%	7%	11%

**Table 7. Peak performance for all configurations.**

do not make a difference here).

The speed up improvement when realistic fetch is taken into account is two to three times larger than perfect fetch. Although the two measurements are not directly comparable because one baseline uses realistic fetch and one uses perfect fetch, the larger speed up of realistic fetch demonstrates that the impact of code realignment and trace caching provides large improvements.

Perfect fetch peak provides improvements roughly corresponding to arbitrary blocking selection at trace length 10. The speed ups due to optimization at trace lengths of 1000 are roughly double that demonstrated by perfect fetch peak, which demonstrates that a lot of optimization opportunity remains. Additionally, roughly double the number of instructions can be removed for arbitrary blocking with 1000 block traces. Tree height reduction, although not as significant, still has a lot of opportunity. In summary, optimization potential can be increased significantly beyond what the evaluated trace selectors provide.

## 6 Related Work

There has been sizable activity in research and development for runtime systems, not only for the benefit of optimization, but also to compile intermediate byte code [1], and for translation of one form of low-level code into another, such as the translation of one ISA into a host ISA [9, 11, 17, 19, 23]. For just-in-time compilation and binary translation, dynamic optimization is one component of the process. For such systems, the benefit from dynamic optimization may be even more substantial due to the removal of the costly interpretation that is required otherwise.

In this paper, we focus primarily on optimization potential of trace-based dynamic optimization systems, and many examples of real systems now exist [2, 4, 6, 8]. Work in dynamic optimization extends further than these software systems, as dynamic optimization support can be added into the hardware layer [13, 16, 21, 25]. The results presented in this paper highlight the performance potential of such systems, and introduce optimization factors that can serve to improve the performance of such systems.

Several researchers have devised techniques for effective trace selection and code caching both in the software layer [3, 5, 10, 18, 24] and for hardware trace caching [13, 28, 29]. Many of these previous works evaluate the

coverage, reuse, and trace lengths of specific schemes for trace selection and caching. Our work examines the coupling between coverage, performance, and trace cache size.

## 7 Conclusions

While the area of dynamic optimization has been heavily explored, the limits of its potential are not well understood. This is primarily due to the difficulty in extracting overhead-free measurements in real optimization systems, and because implementation complexities and design decisions in real systems create limitations that make ideal studies infeasible.

Using an optimizer and performance simulator, we provide an estimation of performance potential for trace-based dynamic optimizers that perform optimizations common to current dynamic optimization systems. When examining performance potential, we found that exponential increases in trace length produce diminishing returns for performance, tree height, and instruction count, ultimately limiting speed up to under 2x. Atomic trace optimization was shown to be much better than non-atomic, which means that, for performance, trace-based optimizers should either have hardware support for atomic traces or use trace scheduling or superblock scheduling to approach atomic optimization performance. Ambiguous memory accesses limit the effectiveness of optimization, and larger improvements are possible for dynamic optimizers that are able to disambiguate or correctly guess memory dependence relationships. Throughout our experiments, we demonstrated much better optimization performance for SPARC than x86. We also explained that SPARC obtains tree height reduction and instruction count reduction benefits from x86 memory addressing modes and larger constant values. Our results demonstrate that it may be beneficial to translate SPARC instructions to micro-instructions, even though it is already a RISC ISA.

In our second experiment, we evaluated performance, coverage, and trace cache size for various trace selection algorithms. We found linear coverage and performance increases require exponential increases in the number of instructions to be optimized. This finding implies that powerful dynamic optimizers need to be efficient, because overhead becomes increasingly more difficult to recoup due to the exponential increase in instructions to be optimized. Next-Executing Tail (NET) trace selection algorithm was shown to be cost-effective for attaining performance, coverage, and trace cache size trends similar to those of higher overhead algorithms, but is limited to producing smaller code sizes, which result in lower performance. Algorithms currently in use in dynamic optimizers today are effective and approach or achieve the same performance, coverage, and trace cache size as an oracular algorithm. We also found that partial trace matching is important to attaining high performance.

Last, we demonstrate that selection algorithms only achieve roughly half of the optimization potential available for longer trace lengths.

## 8 Acknowledgements

We thank the other members of the Advanced Computing Systems group. This material is based upon work supported by the National Science Foundation under Grant Nos. 0092740 and 9984492 with very gracious support

## References

- [1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, 1988.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical Report HPL-1999-78, Hewlett-Packard Laboratories, June 1999.
- [3] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, 2003.
- [4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 265–275, 2003.
- [5] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 79–90, 2003.
- [6] W. Chen, S. Lerner, R. Chaiken, and D. Gilles. Mojo: A Dynamic Optimization System. In *3rd Workshop on Feedback-Directed and Dynamic Optimization*, 2000.
- [7] R. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.
- [8] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Redstone: An On-line Program Specializer. In *In Proceedings of Hot Chips XI*, 1999.
- [9] 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. In *Proceedings of the 1st International Symposium on Code Generation and Optimization*, pages 15–24, 2003.
- [10] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
- [11] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, 1997.
- [12] K. Ebcioglu, E. R. Altman, M. Gschwind, and S. Sathaye. Optimizations and oracle parallelism with dynamic translation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 284–295, 1999.
- [13] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. A performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th Annual International Symposium on Microarchitectre*, 2001.
- [14] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. Continuous optimization. Technical Report UILU-ENG-04-2207, University of Illinois at Urbana-Champaign, August 2004.
- [15] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.
- [16] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 173–181, 1998.
- [17] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33:54–59, Mar. 2000.

- [18] K. Hazelwood and M. D. Smith. Generational cache management of code traces in dynamic optimization systems. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.
- [19] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3 – 12, 1997.
- [20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(9-50), 1993.
- [21] Q. Jacobson and J. E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1999.
- [22] M. Jennings, H. Zhou, and T. Conte. A Tregion-based Unified Approach to Speculation and Predication in Global Instruction Scheduling. Technical report, Dept. of Electrical and Computer Engineering, North Carolina State University, 2001.
- [23] A. Klaiber. The technology behind Crusoe processors. Transmeta White Paper, Jan. 2000.
- [24] M. C. Merten, A. R. Trick, C. N. George, J. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.
- [25] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.
- [26] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 303–313, 2000.
- [27] R. Pearl and L. J. Reed. On the rate of growth of the population of the united states since 1790 and its mathematical representation. *Mathematical Demography*, 1977.
- [28] R. Rosner, M. Micha, Y. Sazeides, and R. Ronen. Selecting long atomic traces for high coverage. In *Proceedings of the 17th International Conference on Supercomputing*, 2003.
- [29] B. Slechta, D. Crowe, B. Fahs, M. Fertig, G. Muthler, J. Quek, F. Spadini, S. J. Patel, , and S. S. Lumetta. Dynamic optimization of micro-operations. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, 2003.
- [30] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, Nov. 2002.