Evaluating the Impact of Dynamic Binary Translation Systems on Hardware Cache Performance

Arkaitz Ruiz-Alvarez Kim Hazelwood
University of Virginia

Abstract

Dynamic binary translation systems enable a wide range of applications such as program instrumentation, optimization, and security. DBTs use a software code cache to store previously translated instructions. The code layout in the code cache greatly differs from the code layout of the original program. This paper provides an exhaustive analysis of the performance of the instruction/trace cache and other structures of the microarchitecture while executing DBTs that focus on program instrumentation, such as DynamoRIO and Pin.

We performed our evaluation along two axes. First, we directly accessed the hardware performance counters to determine actual cache miss counts. Second, we used simulation to analyze the spatial locality of the translated application. Our results show that when executing an application under the control of Pin or DynamoRIO, the icache miss counts actually increase over 2X. Surprisingly, the L2 cache and the L1 data cache show a much lower performance degradation or even break even with the native application. We also found that overall performance degradations are due to the instructions added by the DBT itself, and that these extra instructions outweigh any possible spatial locality benefits exhibited in the code cache. Our observations held regardless of the trace length, code cache size, or the presence of a hardware trace cache. These results provide a better understanding of the efficiency of current instrumentation tools and their effects on instruction/trace cache performance and other structures of the microarchitecture.

1. Introduction

Dynamic binary translation offers a new perspective for program analysis and modification. During the last few years, researchers have developed several dynamic binary translation systems, or DBTs, as extensible execution environments. Examples of these DBTs are Dynamo [2] and DynamoRIO [5, 6], Valgrind [19], Strata [25] and Pin [16]. The most common applications are program instrumentation [16], dynamic optimization [2] and secure execution [13]. There are two main methods for implementing binary instrumentation: probe-based and JIT-based approaches. In probe-based systems, the tool adds trampolines in the original binary to jump to the payload code. In JIT-based systems, the DBT uses a JIT compiler to dynamically modify and cache a copy of the program. The modified copy of the program is divided into fragments and stored in a code cache. In this paper, we focus on JIT-based systems whose main focus is program instrumentation. We use the term *dynamic translation systems*, or DBTs, to refer to tools such as Pin, DynamoRIO, Valgrind, or Strata. We use the term *translated program* to refer to programs running under the control of a DBT with no added instrumentation. Finally, we focus on DBTs that provide an API for performing program instrumentation: Pin and DynamoRIO.

Every DBT adds overhead to the execution of the translated program. This fact is particularly problematic for DBTs that focus on improving performance via program optimization, such as Dynamo [2]. For other translation systems that focus on program instrumentation, such as Pin and DynamoRIO, performance still remains an important issue in that it may restrict the applicability of the system. Therefore, several optimizations have been developed to help improve the performance of DBT systems, such as introducing a software-based code cache to store previously translated code [10]. However, little attention has been paid to the impact of the code cache on the underlying hardware-based instruction cache of the processor. As the DBT changes the program layout, one should expect to see significantly different cache behavior. Furthermore, other researchers have presented benchmarks in which the translated binary performs as well or better than the original binary [2, 8]. The explanation provided for this phenomenon is the *icache effects* – the performance improvement of the processor's instruction cache due to better program locality. We can find examples in the literature [3, 18, 24] that support this hypothesis. Bala et al. [2] specifically mention that one "performance opportunity is instruction cache utilization, since a dynamically contiguous sequence of frequently executing instructions may often be statically non-contiguous in the application binary." However, we have not been able to find instruction/trace cache benefits while using DBTs whose main purpose is program instrumentation (except for one benchmark, `hmmer`, executing under the control of DynamoRIO).

We are not aware of any work that has provided an in-depth analysis of hardware cache behavior of applications running under the control of DBTs. Thus, in this paper, we provide a thorough evaluation of the cache performance of dynamically translated programs. The goals of our work are as follows:

- To measure the effect of dynamic binary translation on the system cache (instruction/trace cache and L2 unified cache) using hardware performance counters.
- To investigate locality changes within the code cache.

978-1-4244-2778-9/08/$25.00 ©2008 IEEE

- To analyze how changes in the trace size, the code cache size, and the use of instrumentation influence the performance of dynamic translated programs.
- To correlate these measurements to the near-native performance of certain benchmarks running under Pin and DynamoRIO.

Our measurements, using the SPECint2006 benchmarks, show that trace cache miss counts increase by 248% over native with Pin, and 170% over native for DynamoRIO. This increase in instruction/trace cache miss counts is exhibited even in benchmarks that perform similar to native overall. These miss counts increase further when the DBT has a limited space for the code cache (a limitation that often appears in embedded systems) and it is not able to fit the entire working set of the benchmark in the code cache. I-TLB performance is also affected by dynamic translation systems – the number of I-TLB misses is increased for every benchmark. The effect of dynamic binary translation is lower in the L2 cache, with an increase in the number of misses by 12% for Pin and 24% for DynamoRIO. The L1 data cache, on the other hand, performs very similar to a native run, even with the added overhead of the DBT's data structures.

In order to understand these significant changes in performance, we visualize the locality of translated binaries by dynamically analyzing the location of the instructions executed in the code cache and in the original program. Our experiments involve two microprocessors – an Intel Core 2 Xeon E5310 (which has a traditional instruction cache) and a Pentium4 (which has a trace cache). Our experiments show that the major factor that affects overall performance is the increased number of executed instructions, and that the instruction (or trace) cache miss rates, L2 cache miss rates, trace size or trace layout do not correlate to the performance of the entire system.

2. Background

Many researchers have developed dynamic binary translation frameworks in recent years. Dynamo [2] is a DBT for PA-RISC whose original focus was program optimization. DynamoRIO [6] is an evolution of Dynamo (for x86) that offers a user-interface for implementing custom modifications. Other alternatives are Pin [16], Valgrind [19] and Strata [25]. In this paper, we use Pin and DynamoRIO to perform the evaluation. Thus we focus on DBTs whose main purpose is program instrumentation.

Pin is an instrumentation system designed and implemented at Intel that provides a cross-platform API. Pin's latest version runs on three architectures: IA32 (32-bit x86), IA32e (64-bit x86), and Itanium; and three operating systems: Windows, Linux and MacOS (x86 version). Users may use Pin's API to implement portable plug-in tools for program analysis, which are called *Pintools*. A Pintool consists of instrumentation routines and analysis routines. *Instrumentation routines* determine where Pin should call the analysis routines. *Analysis routines* perform predetermined actions and have complete visibility of the program and the architectural state. This API provides a simple, transparent way to investigate the behavior of programs at run-time.

To translate an executable binary, Pin attaches to the application like a debugger. Once Pin is injected into the application, it uses a just-in-time compiler to translate and instrument the program, while retaining control of it. The unit of compilation in Pin is the *trace*. A trace is a sequence of instructions that ends in an unconditional control transfer, a pre-defined number of conditional control transfers, or a predefined number of instructions (e.g. 70). After Pin has selected a new trace, the Pintool adds instrumentation using the API function calls, and Pin compiles the instrumented trace and inserts it in a code cache. Thus, none of the original program instructions are executed as Pin only executes instrumented code in the code cache.

DynamoRIO is another DBT implemented in Windows and Linux for IA32. DynamoRIO's functionality is similar to Pin's and it exports an API for developing DynamoRIO clients. A DynamoRIO client consists of hook functions that are called by DynamoRIO when translating the original program (analogous to a Pintool). Through the use of these functions, the user may inspect and modify the program.

Similar to Pin, DynamoRIO must inject itself into the application to gain control and perform the modifications using a JIT compiler. However, the trace formation algorithm differs between DynamoRIO and Pin. DynamoRIO applies the *Next-Executed Tail* [8] heuristic, while Pin forms traces from fall-through paths. In addition, DynamoRIO only forms traces from frequently-executed (50 executions) basic blocks. Thus, DynamoRIO keeps two separate caches of code in software: a *basic block cache* and a *trace cache*, while Pin has a single, unified trace cache.

In this paper, we evaluate the effects on locality of translated programs without instrumentation using Pin to compare the locality of the original program and Pin's code cache. We also perform a comparison of cache performance of Pin, DynamoRIO, and the original program using hardware performance counters.

3. Experimental Setup

Our experimental setup involves two microprocessors, an Intel Pentium4 (32-bit with a trace cache) and an Intel Xeon Core 2 (64-bit with a traditional icache). For physical hardware measurements, we use the interface provided by `PAPI` [15] and `perfex` to read hardware performance counters. The hardware counters report aggregate data, not specific address patterns, therefore we used simulation to evaluate the *spatial locality* of the instructions. To this end, we implemented a Pintool that gathers data about the relative location of consecutive memory references of an application running under Pin's control and natively. We felt that this combination of hardware measurements with instrumentation-based data offered a more complete view of the impact of DBTs on cache performance.

3.1 Hardware Performance Counter Measurements

We used the hardware performance counters to gather certain statistics: the number of accesses/misses in the trace

Papiex Event	Description
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_TLB_IM	I-TLB misses
PAPI_BR_MSP	Cond. branches mispredicted
PAPI_BR_PRC	Cond. branches correctly predicted
PAPI_TOT_INS	Instructions completed
PAPI_RES_STL	Cycles stalled on any resource
PAPI_TOT_CYC	Total cycles
PAPI_L1_ICA	Level 1 instruction cache accesses
PAPI_L2_TCA	Level 2 total cache accesses
Perfex Event	**Description**
0x410143	L1 data reads and writes
0x410F45	L1 data cache lines replacements
0x410047	Weighted L1 miss cycles outstanding
0x410080	Fetches from ICache, stream buffers
0x410081	Fetch miss from ICache, stream buffers
0x410086	Cycles IFU stalled waiting for memory
0x410089	Branches mispredicted at execution
tsc	Total cycles of program execution

Figure 1. List of events measured using the hardware performance counters. `papiex` was used on the Pentium4 32-bit processor, and `perfex` on the Xeon Core 2 64-bit.

cache (Pentium4) and instruction cache (Core2), L2 cache accesses/misses, I-TLB misses, branch prediction accuracy, cycles spent executing the program, and the total number of instructions executed. For each program, we compare the performance of the original binary with the performance under both Pin and DynamoRIO with no instrumentation added and no limit on the code cache. We used the SPECint2006 benchmarks with reference inputs and three iterations. All graphs include error bars that show an 80% confidence interval on the sample average over the various repetitions and inputs (for benchmarks with more than one reference input).

Both the Pentium4 and Core2 systems run a Linux kernel 2.6.9 (i686 for 32-bit and x86_64 for 64-bit) with the *perfctr* patch applied. The 32-bit machine is a Pentium4 Xeon 3.20 GHz Dual Core running CentOS 4.6 with 8 GB of RAM. The Pentium4 includes a 12 K-uop, 8-way associative trace cache. The L1 data cache is a 16 Kb, 8-way associative cache with 64-byte lines. The unified L2 cache is a 2 Mb, 8-way associative cache with 64-byte lines.

The 64-bit machine is a Xeon E5310 QuadCore (Intel Core 2 architecture) with 8GB of RAM running CentOS 4.6. The E5310 includes a traditional instruction cache of 32 Kb, 8-way associative with 64-byte cache lines. The L1 data cache also has 32 Kb, 8-way associativity and 64-byte cache lines. The size of the L2 unified cache is 4 Mb, and it is 16-way associative with 64-byte cache lines.

In order to read the hardware performance counters, we implemented two simple plug-in tools (a *Pintool* and a *DynamoRIO client*) that use the PAPI interface. Both plug-in tools initialize the hardware performance counters at the beginning of the program and print their values when the application completes. These lightweight tools do not have a measurable impact on the performance of either Pin or DynamoRIO. The native performance data was obtained using `papiex`, which is a simple tool that takes a program as the input, executes it and prints out the final hardware performance counter values when the program completes. We did not use `papiex` for gathering statistics about Pin and DynamoRIO since they interfere at launch time.

For the 64-bit platform (Intel Core 2) we used the `perfex` tool due to the documented errors in the way `papiex` measures L1 cache misses on the Core2. Perfex is a lower-level tool that provides a superset of the functionality of `papiex`. We were able to use `perfex` to run the native and translated programs under Pin's control. We present only results based on Pin since DynamoRIO does not operate on a 64-bit operating system. Table 1 shows the specific events measured with both PAPI and `perfex`.

3.2 Simulated Hardware Measurements

We use simulated hardware to investigate the changes in the spatial locality of the application. We classify each executed instruction into one of three different groups. An instruction which is stored in the same cache line as the last executed instruction is classified as *same cache line*. Instructions that are preceded by an instruction on the same page but with a different cache tag are classified as *same page*. Finally, instructions that are preceded by an instruction on another page are counted in the *different page* group. We assign the following parameters to our simulated memory system: a cache line size of 32 bytes and a page size of 4 Kb. These measurements are not intended to be interpreted as a direct translation into the instruction cache and I-TLB hit/miss ratio (although they are correlated). They are collected to instead provide an intuitive indication of the spatial locality exhibited by the application.

Our Pintool keeps track of two different addresses: the original program address and the translated code cache address. Thus, for every run we collect the following data:

- We gather original program addresses to obtain information about the spatial locality of the original application. Although Pin and DynamoRIO never execute the original program, both systems will tell us what original addresses would have executed, which we can use to drive our simulator. Thus, we are able to obtain a fair locality comparison since both programs, original and translated, share the exact same execution path.
- We gather code cache addresses to obtain information about the spatial locality of Pin's code cache. However, function calls to the analysis routines are present in the code cache, and may potentially affect the measurements. We have tried to minimize the impact of the these routines on the collected data since we know the point at which we are inserting the call to the analysis routine (at the beginning of each basic block) and the approximate size in instructions (2 to 3) of this call.

The instrumentation is done at a basic block granularity. For each block in the code cache we keep a list of the instruction memory address references of the original program and the translated program The translated program instructions include those added by Pin to correctly execute the translated program under its control.

When the basic block is executed, Pin calls an analysis routine that performs the following operations:

- Classifies the first instruction of the basic block by comparing its address to the address of the last instruction executed (this address is a global variable).
- Adds the number of instructions in each category to the general counters.
- Updates the global variables with the addresses of the last instruction of the basic block.

At program termination, the statistics are stored on disk. Therefore, the Pintool simulates in each run two different streams of memory references based on the original program instructions and the translated program instructions. The main advantage of the simulation is that we need not factor in the impact of other Pin modules, such as the compiler and dispatcher, and can focus exclusively on the spatial locality of the native versus translated application.

For our simulations, we again used SPECint2006, but for the benchmarks with more than one reference input, we ran only the first input since the simulation process is orders of magnitude slower than native. We ran each benchmark three times, each time with a different value of the maximum trace size allowed in Pin's code cache. These values are 70 (the default in Pin), 40 and 15 instructions. We also compared the differences between 32-bit (Pentium4) and 64-bit (Intel Core 2) systems.

4. Evaluation

We begin by reporting the measurements obtained using hardware performance counters. These measurements include several structures of the microarchitecture: the instruction/trace cache, I-TLB, L2 cache, and branch predictor. We show the results when Pin and DynamoRIO have no limit in code cache size and add no instrumentation. We also comment on our observations when we vary these two parameters. Next, we perform a comparison of the locality exhibited by translated binaries. We use the Pintool described in Section 3.2 to gain insight on the impact of the DBT on the spatial locality of the application.

4.1 Hardware Performance Counters

As described in Section 3.1, we measured the cache performance of binaries running under Pin and DynamoRIO. First, Figure 2 shows the running time (cycles) for each benchmark on a 32-bit architecture. The benchmarks are ordered from shortest run time running under Pin – `mcf` – to longest run time – `perlbench`. As we can see, several benchmarks perform very close to native: `mcf`, `libquantum`, `hmmer` and `bzip2`. Figure 3 shows similar results for a 64-bit architecture, this time including the number of instructions executed. On a 64-bit machine, only `mcf` is able to perform close to native with a slowdown of 5%.

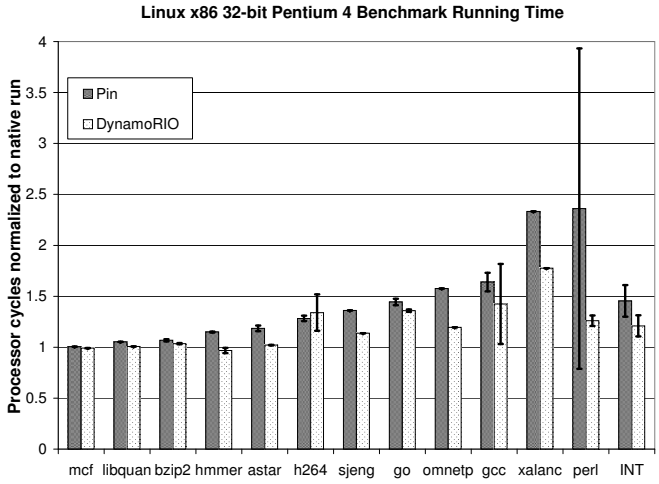

Figure 2. Duration (cycles) of each SPECint2006 benchmark while running under DynamoRIO and Pin. Results are normalized to native run, and are sorted fastest to slowest.

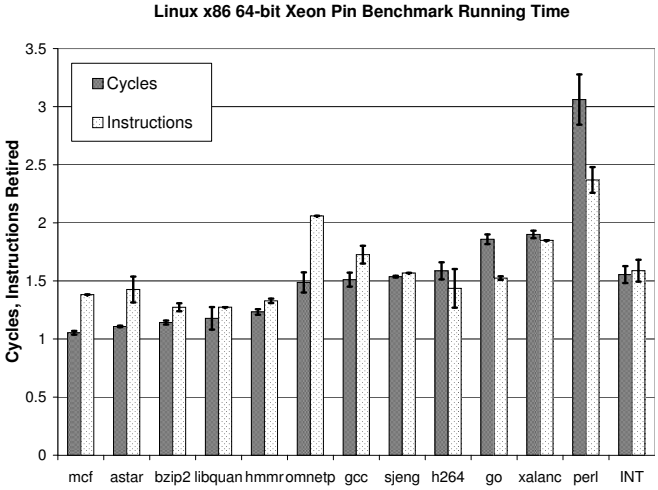

Figure 3. Duration of each SPECint2006 benchmark (cycles and instructions), running under Pin. Results are normalized to native run, and are sorted fastest to slowest.

Level 1 Instruction and Trace Cache Figure 4 shows the normalized trace cache miss and access counts for the SPECint2006 benchmarks on a 32-bit machine. The benchmarks are ordered by their running time (shown in Figures 2 and 3). The benchmarks `libquantum` and `hmmer` running with DynamoRIO represent effective trace cache behavior compared to native execution (34% fewer trace cache misses for `libquantum` and 15% fewer for `hmmer`). However, these two benchmarks are the exception rather than the norm. All the

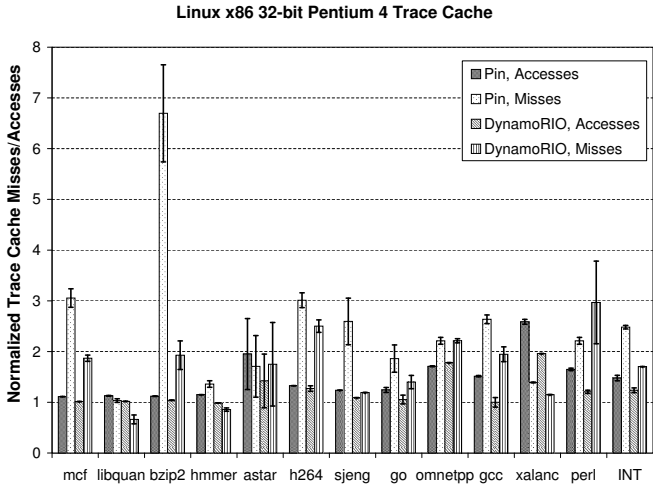

Figure 4. Access and miss counts in the Pentium4 trace cache for the SPECint2006 benchmarks with Pin and DynamoRIO. Results are normalized to native (lower is better).

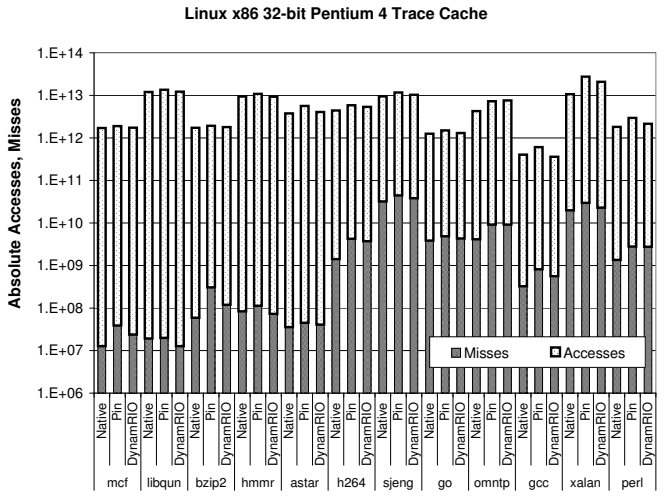

Figure 6. Pentium4 trace cache accesses and misses for the SPECint2006 benchmarks with Pin and DynamoRIO. The y-axis has a logarithmic scale (lower is better).

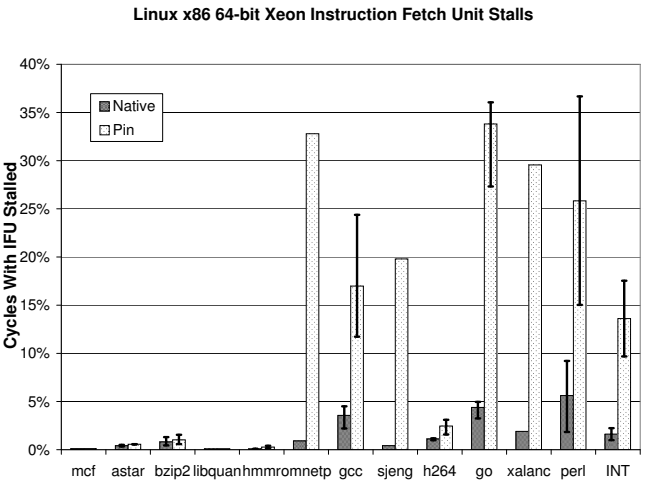

Figure 5. Percentage of cycles during the SPECint2006 benchmark execution in which the Xeon E5310 instruction fetch unit is stalled waiting for data from memory (lower is better). Benchmarks were run under Pin.

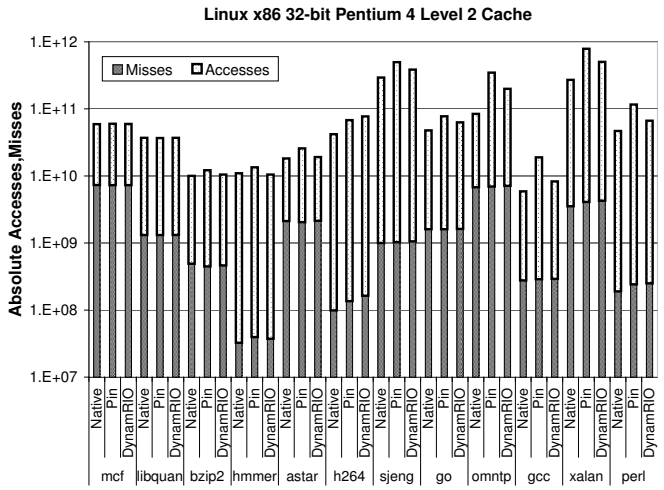

Figure 7. Pentium4 L2 cache accesses and misses for the SPECint2006 benchmarks with Pin and DynamoRIO. The y-axis has a logarithmic scale (lower is better).

other benchmarks perform similar or worse, with an average of 2.5X more misses for Pin and 1.7X more for DynamoRIO. The performance of the trace cache is closely related to the I-TLB, which also increases in misses for every benchmark. Figure 6 also presents the raw, non-normalized performance, showing that the significant increase in trace cache misses is not a product of low miss counts in the native run. If we only consider benchmarks whose native trace cache miss count is close to or greater than one billion, the increase in trace cache misses under Pin is still at 230%.

We also performed the same experiments on the 64-bit architecture. However, the low number of L1 instruction cache misses greatly distorts the numbers when we normalize them to native performance. Many benchmarks go from a very low number of instruction cache misses (on the order of 100K) to a dramatic increase (240x) without affecting the performance significantly. Therefore, we explored other metrics. Figure 5 shows the percentage of the execution time (in cycles) in which the processor's instruction fetch unit (IFU) is waiting for data from memory. There is no benchmark that shows an improvement in instruction cache behavior for any metric: absolute number of misses or absolute number of cycles in which the IFU is stalled. Furthermore, most translated benchmarks that perform poorly compared to native show a significant increase in the number of cycles in which the IFU cannot fetch new instructions.

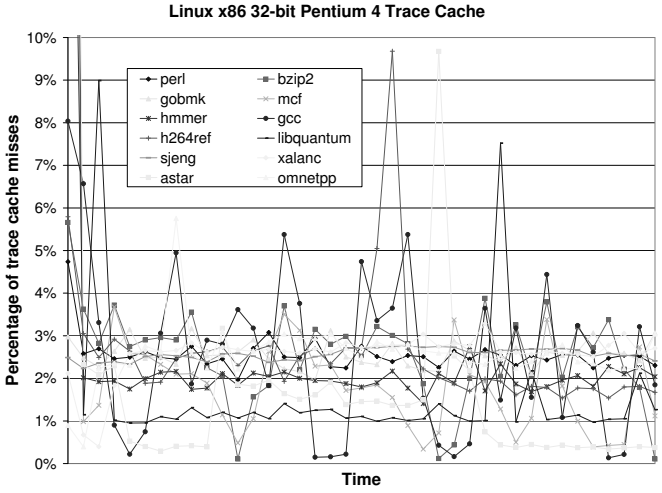

Figure 8. Distributions of Pentium4 trace cache misses over time for the SPECint2006 benchmarks. Time is normalized for each benchmark: data points were gathered at every 2.5% of the duration of the benchmark.

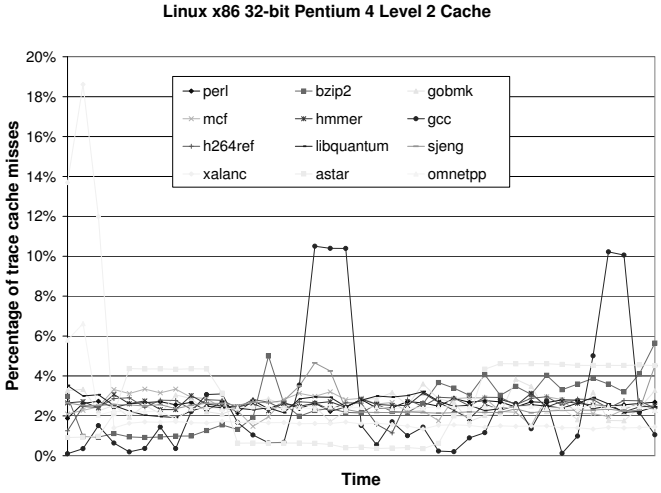

Figure 9. Distributions of Pentium4 L2 cache misses over time for the SPECint2006 benchmarks. Time is normalized for each benchmark: data points were gathered at every 2.5% of the duration of the benchmark.

Benchmarks that perform well (total running time) show a significant increase in trace cache or instruction cache misses. This increase may be caused by poor code layout, a bigger memory footprint, or a greater number of executed instructions. When a DBT compiles a trace or a basic block, it must add instructions to ensure the correct behavior of the application: spilling registers, adding exit stubs, resolving conditional branches, etc. As we will later show, a translated binary executes many times more instructions than the native binary, and this is the main factor for this poor trace cache performance. Additionally, the binary image is bigger for translated programs since we now have to include the DBT in the memory footprint. Thus, benchmarks that used to fit in the L1 instruction cache will no longer fit when run under a DBT. The DBT's routines compete with the application for cache space, degrading its performance.

Figures 8 and 9 show the distribution of trace cache and L2 misses over time for the SPEC benchmarks. Many misses do occur at the beginning of the execution, and they are relatively stable for the remainder of execution, with the exception of phase changes for benchmarks like `gcc`. Yet, the misses are high enough for many applications throughout run time that its clear that we do not quite reach a *steady state* where only translated code is executing.

We repeated these runs varying the code cache size and adding instrumentation, and we summarize our results. If we limit the code cache size, the performance of many benchmarks completely degenerates as expected, with a miss count two orders of magnitude worse than the native run. We found that frequent code cache flushing sharply increases the miss count. For benchmarks with a working set that does not fit in the code cache, this is a serious performance issue. There are situations in which the DBT has a limited amount of memory space available, either because the device is embedded, or because a large, commercial application is monopolizing all of the resources.

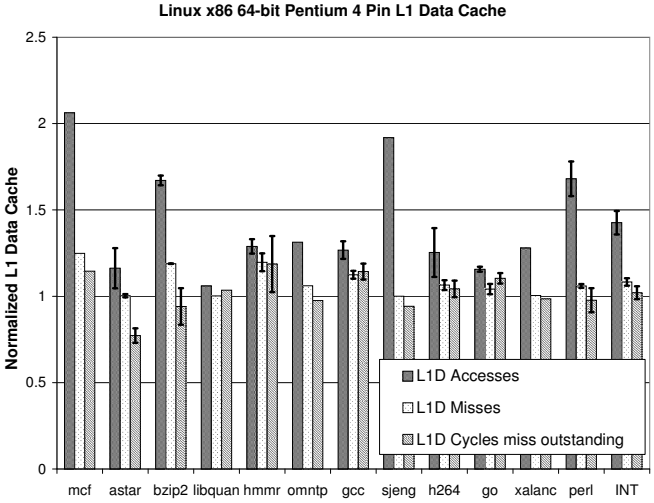

Figure 10. Access and miss counts (lower is better) for the L1 data cache of the the Xeon E5310, including cycles during which the L1 data cache has an outstanding miss. Benchmarks are running under Pin's control.

When we include instrumentation, there is also a sharp increase in the miss counts. We tried adding instruction-level instrumentation (inlined), and trace-level instrumentation (inlined and not inlined). The DBT greatly slows down the application and, as expected, the performance of the trace cache, L2 cache, I-TLB, and branch predictor degenerates.

Level 1 Data Cache In Figure 10, we show the L1 data cache performance on the 64-bit Core2 machine. This figure includes the accesses, misses, and cycles in which the L1 data cache has

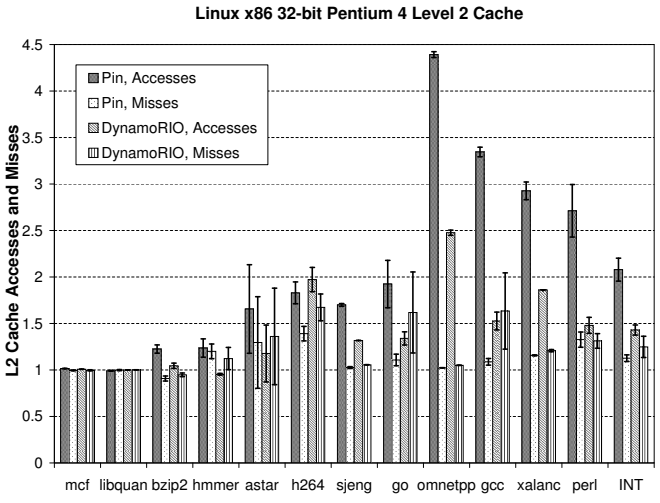

Figure 11. Access and miss counts in the Pentium4 L2 cache for the SPECint2006 benchmarks with Pin and DynamoRIO. Results are normalized to native (lower is better).

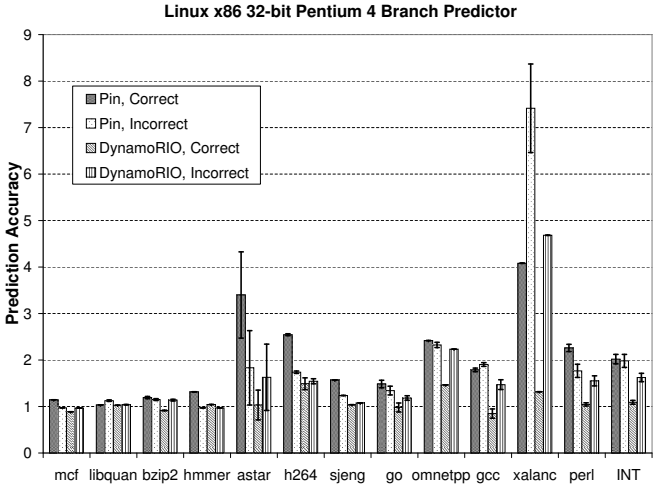

Figure 12. Pentium4 branch predictor performance. We report the number of correct and incorrect predictions while running SPECint2006 under DynamoRIO and Pin. Results are normalized to a native run.

an outstanding miss. Running a program under Pin puts additional pressure on the data cache: Pin's data structures used during the JIT compiling process and the bookkeeping (to retain control of the translated program) cause an increase in the number of accesses to the data cache. However, we can see that the relative number of data cache misses is very close to native. This surprising result indicates that the locality of Pin's data structures do not negatively impact the L1 data cache performance. However, since the number of misses and cycles in which the data cache is waiting for data from memory do not significantly improve over the native run we can not claim that performance benefits will arise from this superior data layout. We can conclude that, regarding the L1 data cache performance, both the native program and the translated program break even.

Level 2 Cache Figure 11 shows the performance of the L2 unified instruction/data cache. Despite the large increase in trace cache misses when running with Pin, the L2 cache is able to perform reasonably well compared to the original program. Figure 7 also presents the raw, non-normalized performance. We observed an average increase of 20-25% for all the INT benchmarks, and found that half of the benchmarks perform as well as the original program. Pin and DynamoRIO build fragments of code at run-time as the program execution advances, so code that is temporally near at execution time should be spatially closer in the code cache than in the original binary image. It may be the case that the instructions added by the DBT cause the trace cache to have a higher miss rate, but the superior code cache organization causes the L2 cache performance to remain consistent.

Although we do not include an explicit graph, we found that the picture is very similar on a 64-bit machine. All benchmarks, except `gcc`, perform very close to native in L2 misses, with an average of a 8% increase in the number of misses if we

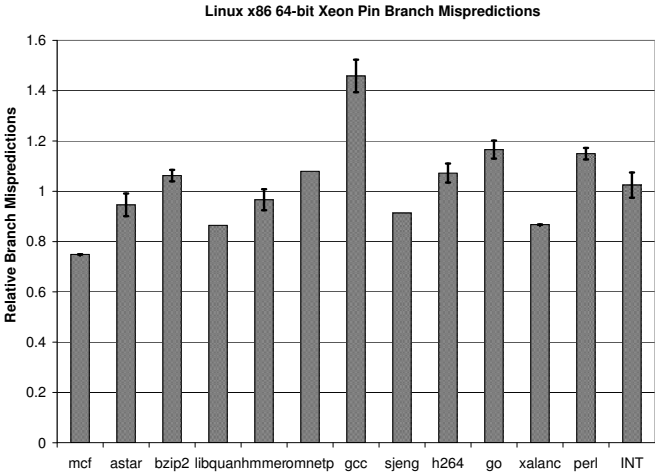

Figure 13. Number of branch instructions executed and mispredicted at execution for the SPECint2006 benchmarks. Results are normalized to a native run.

exclude `gcc`. This is a surprising result given that the increase in the number of L1 instruction cache misses puts additional pressure on the L2 cache. This result is similar to the L1 data cache findings: both the native and translated program have comparable misses even with a greater number of accesses to both caches. However, this improved code and data layout is not enough for the translated program to reap performance benefits overall.

Branch Prediction We present the branch predictor performance in Figure 12 for the Pentium4 and Figure 13 for the Core2. There are significant increases in the number of mispredictions for some benchmarks, mainly those with the high-

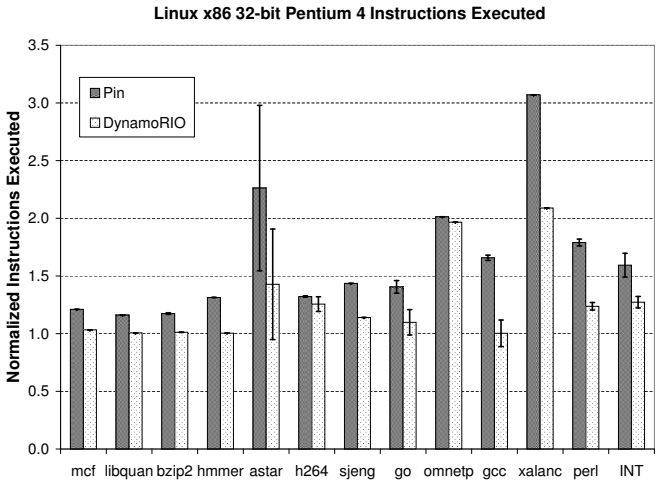

Figure 14. Instructions executed for each SPECint2006 benchmark running under DynamoRIO and Pin. Results are normalized to a native run.

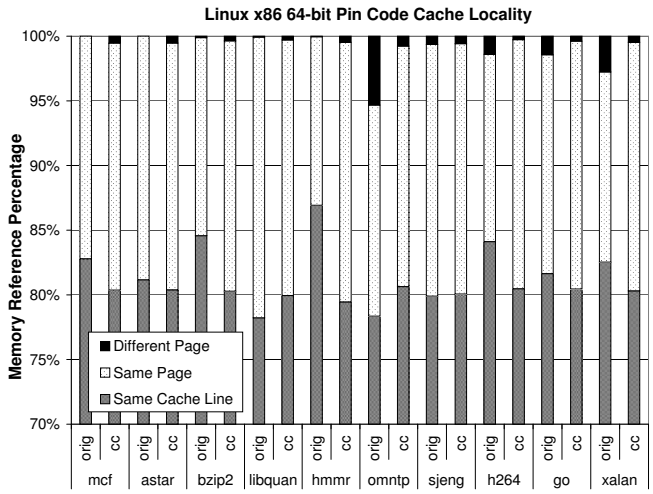

Figure 15. Percentage of instruction memory references for each of the following groups: next instruction is stored in the same cache line, next instruction is on the same page, next instruction is on a different page. We compare the original program instructions to Pin's code cache.

est slowdowns when running under Pin or DynamoRIO. Some benchmarks have a similar number of mispredictions running under Pin or DynamoRIO compared to a native run. A common characteristic of these benchmarks (`mcf`, `libquantum` and `bzip2` for Pentium4 and `mcf` for Core2) is that the running time of the translated binary is close to the running time of the native binary. The benchmarks with no significant increase in mispredictions correlate to those with better performance. Previous results [12] have suggested a strong correlation between the handling of indirect branches and the total slowdown exhibited by the translated application.

There are two challenges to achieving similar branch prediction performance in a translated program. First, resolving the target of an indirect branch is a costly process that needs to invoke the DBT's routines. Second, the code cache can compile and duplicate the same original code into several, sparse basic blocks in the code cache. These basic blocks correspond to the same original program instructions, but the hardware branch predictor does not see the correlation. Thus, part of the branch history used to predict the branch in the original program is lost since it is scattered into several translated branch instructions.

Instructions Executed Figure 14 shows the number of executed instructions by Pin and DynamoRIO normalized to the number of instruction executed by the original program. We see that the benchmarks with the lowest overhead – `mcf`, `libquantum`, `bzip2`, `hmmer` – are those with less intervention from the DBT. Other researchers discuss several characteristics that make a program run under the control of a DBT without significant slowdowns: small binary image, low number of indirect branches [12], and long running times. A small binary image means that the DBT has to translate fewer instructions, so the overhead from the JIT process is reduced. A low number of indirect branches reduces the possibility of having to transition from the code cache to the DBT routines to resolve the target address of an indirect branches, which is expensive. Finally, a long running time amortizes the overhead from the initial translation (JIT compilation process). All these causes of overhead directly correlate to an increasing number of executed instructions. Thus, the near native performance that some benchmarks exhibit is a direct result of the light intervention of the DBT (few JIT compilation and indirect branch solving required) and a long running time.

4.2 Simulation

We use the Pintool described in Section 3.2 to visualize the spatial locality of translated binaries. Figure 15 shows the spatial locality of the SPEC benchmarks executed under the control of Pin on x86-64. (We also performed this experiment on x86-32, and saw similar results.) On both architectures, the vast majority of the instructions are followed by an instruction in the same cache line. Furthermore, page changes are not very common.

In Figure 15, for both the original and translated programs, there is a similar distribution of instructions in each category. In relative numbers, there is not a significant difference, although original binaries tend to have a slightly higher percentage of instructions in the same cache line. Thus, the code layout in the code cache is not responsible for the poor instruction cache performance.

We repeated these experiments after reducing the maximum cached trace length in Pin from its default value of 70 instructions. Despite exploring trace length limits of 15 and 40 instructions, we did not observe a significant impact on the locality or number of instructions executed.

5. Related work

Our work relates to the various dynamic binary translation systems that have been developed over the past few years, such as Dynamo [2], DynamoRIO [6], Valgrind [19], Strata [25] and Pin [16]. We have focused on Pin and DynamoRIO in this paper, although we believe that the results will scale to other DBTs that use a JIT compiler to translate code, and store translations in a code cache.

In the original Dynamo paper [2], the authors mention that most of the speedup gained by Dynamo was due to trace selection, which improves code layout. There are several papers where the authors report that programs running under the control of a DBT have run-time performance similar to the original program [16, 29]. Some studies have pointed to the *icache effects* as a benefit of DBTs [2, 3, 18, 24] and the main justification for superior performance when compared to the original program [8]. However, most of the benchmarks analyzed in our study of instrumentation-oriented DBTs exhibit slowdowns compared to native execution. Only two benchmarks show speedups: `hmmer` and `mcf`, while another three (`bzip2`, `libquantum`, and `astar`) break even with the original binary in terms of performance. Two of them – `hmmer` and `libquantum` – exhibit superior cache performance. Thus, the benchmarks `hmmer` and `libquantum` are the only examples of positive *icache effects*. In general, both Pin and DynamoRIO have a significant negative impact on the instruction/trace cache performance.

Cache performance and locality are very important concepts in computer architecture. Thus, in the past decades there have been many studies [9, 26, 27] that have analyzed the performance impact and improved the design of caches. Apart from hardware solutions, software solutions have been proposed to reduce the miss rate of the instruction cache [17] and the data cache [23]. These software optimizations are implemented in compilers [7, 14, 22] or using profile information [20]. A DBT uses compilation techniques to generate the translated traces, but also may collect run-time information similar to profilers.

Several researchers have carried out cache performance measurements using hardware performance counters to analyze the impact of compilation techniques [21], Java virtual machines [4, 28], or profiling techniques [1]. Our study uses the hardware performance counters to assess the impact of dynamic translation on the system cache. This approach presented a key inconvenience in that hardware counters are unable to differentiate the impact of the translated code from DBT code. Thus, we used instrumentation to provide an intuitive picture of the change in spatial locality as a result of the DBT.

Some authors have analyzed the best strategies to form traces in DBTs, by describing several possible implementations and evaluating the performance of each [3, 11, 12, 18, 24]. Our work differs from these efforts in that the fragment construction policies are fixed in Pin and DynamoRIO. Our motivation is to provide a better insight into the impact of DBTs on instruction/trace cache performance independent of the trace-selection policy.

6. Conclusions

Dynamic binary translation is an effective way to modify applications for purposes such as security, program analysis and optimization. In order to modify the program and to maintain control, DBTs use JIT compilers which translate fragments of the original application code, often storing them in a code cache. Since cached code is executed in lieu of the original application, program layout is affected.

In this paper, we have measured and analyzed the impact of Pin and DynamoRIO on hardware cache performance. Contrary to earlier results with DBTs such as Dynamo, we have found that the icache is negatively affected by binary translation systems. For the SPECint2006 benchmarks, Pin and DynamoRIO increase the number of trace/instruction cache misses by an average of 248% and 170%, respectively. This increase in miss rate did not necessarily correlate with run-time performance, however. Our measurements also reflect a large I-TLB miss increase and a less dramatic increase in L2 cache misses – 20% for Pin and 24% for DynamoRIO. Finally, the L1 data cache performance is similar for both the translated and original binary meaning that the DBT's data structures did not add significant pressure on the data cache.

To explain these results, we presented a visualization of the spatial locality of translated binaries by dynamically analyzing the location of the instructions executed. Our experiments on two processors, Xeon Core 2 (instruction cache) and Pentium4 (trace cache), show that the major factor that affects cache performance is the increase in the number of executed instructions in translated programs.

Acknowledgments

Arkaitz Ruiz-Alvarez is funded by the Caja Madrid Foundation. This work was also made possible by NSF CAREER 0747273, NSF CSR 0720803, SRC-GRC 1790.001, monetary donations from Google and Microsoft, and equipment donations from Intel. We would also like to thank our shepherd, Michael Hind, and the anonymous reviewers for their useful feedback on earlier versions of this paper.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4):357–390, November 1997.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Vancouver, BC, Canada, June 2000.

[3] M. Berndl and L. Hendren. Dynamic profiling and trace cache generation. In *First Int'l Symposium on Code Generation and Optimization*, pages 276–285, San Francisco, CA, USA, March 2003.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. *ACM*

SIGMETRICS Performance Evaluation Review, 32(1):25–36, June 2004.

[5] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, Austin, TX, USA, December 2001.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *First Int'l Symposium on Code Generation and Optimization*, pages 265–275, San Francisco, CA, USA, March 2003.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, USA, June 1999.

[8] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *12th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, Cambridge, MA, October 2000.

[9] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *SIGOPS Operating System Review*, 32(5):228–239, December 1998.

[10] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *Transactions on Code Generation and Optimization (TACO)*, 3(3):263–294, September 2006.

[11] D. J. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *38th Annual International Symposium on Microarchitecture*, pages 141–154, Barcelona, Spain, November 2005.

[12] J. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. Childers. Evaluating fragment construction policies for sdt systems. In *2nd Annual Conference on Virtual Execution Environments*, pages 122–132, Ottawa, ON, Canada, June 2006.

[13] W. Hu, J. Hiser, D. Williams, A. Filipi, J. Davidson, D. Evans, J. Knight, A. Nguyen-Tuong, and J. Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *2nd Annual Conference on Virtual Execution Environments*, pages 2–12, Ottawa, ON, Canada, June 2006.

[14] W.-M. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th International Symposium on Computer Architecture*, pages 242–251, Jerusalem, Israel, May 1989.

[15] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-user tools for application performance analysis using hardware counters. In *14th Conference on Parallel and Distributed Computing Systems*, pages 460–465, Richardson, TX, USA, August 2001.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, IL, USA, June 2005.

[17] S. McFarling. Program optimization for instruction caches. In *Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[18] D. Merrill and K. Hazelwood. Trace fragment selection within method-based JVMs. In *4th Annual Conference on Virtual Execution Environments*, pages 41–50, Seattle, WA, USA, March 2008.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, CA, USA, June 2007.

[20] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, NY, USA, June 1990.

[21] R. M. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *Architectural Support for Programming Languages and Operating Systems*, pages 189–198, Boston, MA, USA, October 2004.

[22] A. Ramirez, J. L. Larriba-Pey, and M. Valero. Software trace cache. *IEEE Transactions on Computers*, 54(1):22–35, January 2005.

[23] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, QC, Canada, June 1998.

[24] K. Scott, N. Kumar, B. R. Childers, J. W. Davidson, and M. L. Soffa. Overhead reduction techniques for software dynamic translation. In *18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, USA, April 2004.

[25] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *First Int'l Symposium on Code Generation and Optimization*, pages 36–47, San Francisco, CA, USA, March 2003.

[26] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[27] A. J. Smith. Second bibliography on cache memories. *Computer Architecture News*, 19(4):154–182, June 1991.

[28] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In *USENIX 3rd Virtual Machine Research And Technology Symposium*, pages 57–72, San Jose, CA, USA, May 2004.

[29] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *5th Annual International Symposium on Code Generation and Optimization*, pages 209–217, San Jose, CA, USA, March 2007.