Code Layout Optimizations for Transaction Processing Workloads

Alex Ramirez*, Luiz André Barroso†, Kourosh Gharachorloo†,
Robert Cohn‡, Josep Larriba-Pey*, P. Geoffrey Lowney‡, and Mateo Valero*

*Computer Architecture Department
Universitat Politecnica de Catalunya

†Western Research Laboratory
Compaq Computer Corporation

‡Alpha Development Group
Compaq Computer Corporation

Abstract

Commercial applications such as databases and Web servers constitute the most important market segment for high-performance servers. Among these applications, on-line transaction processing (OLTP) workloads provide a challenging set of requirements for system designs since they often exhibit inefficient executions dominated by a large memory stall component. This behavior arises from large instruction and data footprints and high communication miss rates. A number of recent studies have characterized the behavior of commercial workloads and proposed architectural features to improve their performance. However, there has been little research on the impact of software and compiler-level optimizations for improving the behavior of such workloads.

This paper provides a detailed study of profile-driven compiler optimizations to improve the code layout in commercial workloads with large instruction footprints. Our compiler algorithms are implemented in the context of Spike, an executable optimizer for the Alpha architecture. Our experiments use the Oracle commercial database engine running an OLTP workload, with results generated using both full system simulations and actual runs on Alpha multiprocessors. Our results show that code layout optimizations can provide a major improvement in the instruction cache behavior, providing a 55% to 65% reduction in the application misses for 64-128K caches. Our analysis shows that this improvement primarily arises from longer sequences of consecutively executed instructions and more reuse of cache lines before they are replaced. We also show that the majority of application instruction misses are caused by self-interference. However, code layout optimizations significantly reduce the amount of self-interference, thus elevating the relative importance of interference with operating system code. Finally, we show that better code layout can also provide substantial improvements in the behavior of other memory system components such as the instruction TLB and the unified second-level cache. The overall performance impact of our code layout optimizations is an improvement of 1.33 times in the execution time of our workload.

1 Introduction

Commercial applications such as databases and Web servers constitute the largest and fastest-growing segment of the market for high-performance servers. While applications such as decision support (DSS) and Web index search have been shown to be relatively insensitive to memory system performance [2], a number of recent studies have underscored the radically different behavior of online transaction processing (OLTP) workloads [2, 6, 7, 15, 17, 19, 25]. In general, OLTP workloads lead to inefficient executions with a large memory stall component and present a more challenging set of requirements for processor and memory system design. This behavior arises from large instruction and data footprints and high communication miss rates that are characteristic for such workloads [2]. At the same time, the increasing popularity of electronic commerce on the Web further elevates the importance of achieving good performance on OLTP workloads.

While there have been several recent studies that characterize the behavior of commercial workloads and propose architectural features to enhance their performance (e.g., [3, 16, 23]), there has been little research on software and compiler-level optimizations to improve the behavior of such workloads. One of the distinguishing features of OLTP workloads that is amenable to compiler-level optimizations is their large instruction footprint and the relative importance of the instruction cache performance. Previous studies have shown high instruction cache miss rates for both the application and operating system components. In fact, the number of instruction misses can be higher than the data misses for the same size instruction and data caches [2]. Furthermore, even a reasonably sized second-level unified cache (1MB) can have as high as 50% of its misses due to instructions [2].

This paper provides a detailed study of profile-driven compiler optimizations to improve the code layout in commercial workloads. Our primary goal is to further characterize the instruction cache behavior of OLTP workloads and to study the impact of code layout optimizations on the performance of such workloads. The main compiler algorithms that we study are basic block chaining, fine-grain procedure splitting, and procedure ordering. These optimizations are implemented in the context of Spike [5], an executable optimizer for the Alpha architecture. Spike is a relatively mature optimizer and has been used for generating the more recent audited TPC-C benchmark results for Alpha servers [29]. Our performance results for OLTP are based on executions of the Oracle commercial database engine (version 8.0.4) running under Compaq Tru64 Unix (previously known as Digital Unix). We use a combination of full system simulation (including operating system activity) and actual runs on Alpha multiprocessor platforms for our experiments.

To better understand the behavior of our workload, we begin by studying the application database instruction stream in isolation (achieved by filtering out operating system references). We first characterize the footprint of the application code and study its behavior with different cache parameters. We then analyze the impact of various code layout optimizations on the application code cache behavior. Our results show that code layout optimizations can provide a major improvement in the instruction cache behavior, providing a 55% to 65% reduction in the application misses for 64-128K caches. We find that a line size of 128-bytes is ideal for both the optimized and unoptimized code, and that the improvements from associativity are quite small. The most effective code layout optimization is basic block chaining, with combined procedure splitting and ordering optimizations providing further substantial reductions in instruction misses. Finally, our detailed analysis shows that the instruction cache miss improvements obtained by code layout optimizations can be primarily attributed to (i) longer

0-7695-1162-7/01/$10.00 (C) 2001 IEEE

sequences of consecutively executed instructions and (ii) more reuse of cache lines before they are replaced.

We next study the combined application and operating system instruction streams. The interference between the two streams diminishes the benefits of code layout optimizations, leading to a reduction of 45% to 60% in combined instruction misses (compared to 55%-65% for the isolated application stream) for 64-128K caches. The majority of the application instruction misses are caused by self-interference in the application code. However, code layout optimizations substantially reduce such self-interference, thus increasing the impact of the interference between the application and the operating system. Applying code layout optimizations to the operating system code does not lead to any significant improvements.

Finally, we show that better code layout can also provide substantial improvements in other parts of the memory system such as the instruction TLB and the unified second-level cache. The overall performance impact of the code layout optimizations is an improvement in execution time of 1.33 times across our different simulation and hardware platform experiments.

The rest of the paper is structured as follows. The next section provides an overview of the code layout optimizations used in this study. Section 3 presents our experimental methodology, including a brief description of the OLTP workload. Section 4 characterizes the instruction cache behavior of the database application code and studies the impact of code layout optimizations on the isolated application instruction stream. The combined application and operating system instruction stream is studied in Section 5. Finally, we discuss related work and conclude.

2 Code Layout Optimizations

This section describes the code layout optimizations used in our study. These optimizations are implemented in the context of Spike [5], a Compaq product that optimizes executables for the Alpha architecture. Spike was originally developed for optimizing Alpha NT binaries and was subsequently extended to optimize Tru64 Unix binaries as well. This optimizer has been successfully used to improve the audited results for TPC-C and Oracle application benchmarks running on Alpha servers. Finally, Spike can be used to optimize both user-level applications and operating system code.

A detailed description of the code layout optimizations available in Spike can be found in previous papers [5, 20, 22]. The Spike optimizer algorithm consists of three parts. The basic blocks of a procedure are reordered to sequentialize the most frequent paths. The procedure is then split into segments (or sequences), where each segment ends with an unconditional branch or return. Finally, these segments are mapped into memory using the Pettis and Hansen procedure ordering algorithm[20]. We also implemented a version of the CFA optimization [22], which attempts to reserve a conflict-free area in the instruction cache for the most frequently executed traces. However, the footprint for such traces in our OLTP workload was too large to fit within a reasonably sized fraction of the cache, and the optimization yielded no gains. Therefore, we do not use the CFA optimization in this study.

Our code layout algorithms are profile-driven. Either Pixie (uses instrumentation) or DCPI [1] (uses sampling) can be used for collecting basic block execution counts. Spike then builds the call graph for the entire program and the control flow graphs for each procedure. The call graph includes edges for branches between procedures. A call or branch edge weight is determined by the execution count on the basic block that contains the call. For the flow graph, the control flow edge weights are estimated from the basic block counts.

Basic Block Chaining

Figure 1(a) shows an example of the block chaining algorithm. Spike uses a simple greedy algorithm to order the basic blocks within a procedure. The flow edges are sorted by weight and are processed in order, starting with the edge with the heaviest weight. Each flow edge has a source and destination block. If the edge's source block does not already have a successor and the destination block does not already have a predecessor, the two blocks are chained together. The layout algorithm biases conditional branches to be not taken and eliminates frequently executed unconditional branches. Chaining is complete when all of the edges are processed. At this point, a procedure will consist of one or more chains. The chains are sorted by the execution count on the first basic block. The chain containing the procedure entry point is placed first, and the rest of the chains are placed in decreasing order.

Fine-Grain Procedure Splitting

After the basic blocks in a procedure are chained, we divide the chain into multiple code segments. A code segment is ended by an unconditional branch or return. Each code segment is then represented as a new separate procedure in Spike. The call graph includes branch as well as call edges to represent transitions between these new procedures. Figure 1(b) illustrates our procedure splitting algorithm where a single procedure is split into multiple segments/procedures. The above procedure splitting algorithm was developed for this study and differs from the one currently available in the Spike distribution. The latter algorithm only splits a procedure into a hot and a cold part based on the relative execution frequency of the basic blocks within the procedure.

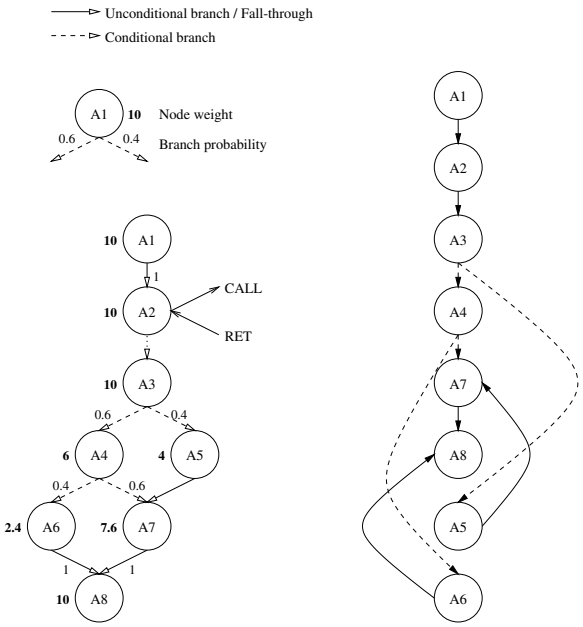
Fine grain procedure splitting leads to a program composed of many segments, each one with a few basic blocks that we expect will execute sequentially. By splitting each procedure into multiple segments/procedures, we get an extra degree of flexibility for the follow-on procedure ordering algorithm.

Procedure Ordering

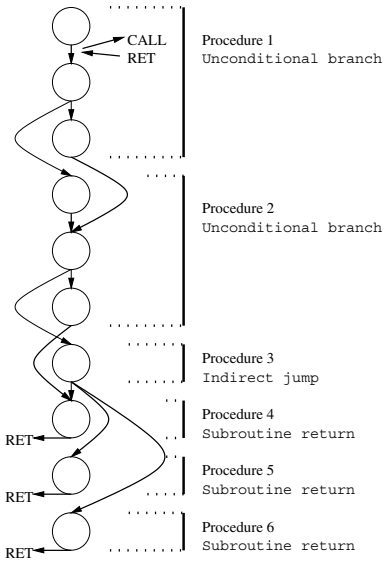
The procedure ordering algorithm is a straightforward implementation of Pettis and Hansen that attempts to place related procedures near one another. The example presented in Figure 2 illustrates the steps. We build a call graph and assign a weight to each edge based on the number of calls. If there is more than one edge with the same source and destination, we compute the sum of the execution counts and delete all but one edge. To place the procedures in the graph, we select the most heavily weighted edge (A to C), record that the two nodes should be placed adjacently, collapse the two nodes into one (A,C), and merge their edges. We again select the most heavily weighted edge and continue until the graph is reduced to a single node (E,D,B,A,C). The final node contains an ordering of all the procedures. When we merge nodes which contain more than one procedure, we use the weights in the original (not merged) graph to determine which of the four possible merge endpoints is best. In addition, special care is taken to ensure that we rarely require a branch to span more than the maximum branch displacement.

3 Methodology

This section describes the OLTP workload, our profiling scheme, and the simulation and multiprocessor hardware platforms used in this study.



(a) basic block chaining algorithm



(b) procedure splitting algorithm

Figure 1: Example of the basic block chaining and procedure splitting algorithms.

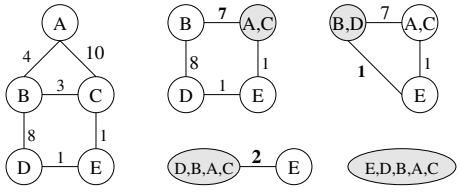


Figure 2: Example of the procedure ordering algorithm.

3.1 OLTP Workload

Our OLTP workload is modeled after the TPC-B benchmark [30]. This benchmark models a banking database system that keeps track of customers' account balances, as well as balances per branch and teller. Each transaction updates a randomly chosen account balance, as well as the balances of the corresponding branch and teller. The transaction also adds an entry to the history table which keeps a record of all submitted transactions. We use the Oracle 8.0.4 commercial database management system as our database engine.

Our OLTP workload is set up and scaled in a similar way to a previous study which validated such scaling [2]. We use a TPC-B database with 40 branches and a size of over 900MB. We use Oracle in a dedicated mode for this workload, whereby each client process has a dedicated server process for serving its transactions. To hide I/O latencies, including the latency of log writes, OLTP runs are usually configured with multiple server processes per processor. We use 8 processes per processor in this study.

3.2 Collecting Profiles

The OLTP profile data was obtained using Pixie. We use the original (unmodified) binary to start up the database, cache all tables in memory, and warm up the indices. In order to focus the profile on the transaction processing component of the workload, only the server processes that are dedicated to executing client requests use the "pixified" binary. The workload is then ran for 2000 transactions after the warmup period.

Kernel profiles were collected using the Tru64 Unix kprofile tool, which is based on PC sampling using the Alpha performance counters. The profile data for the kernel was derived while executing the transaction processing section of the OLTP workload.

3.3 Hardware and Simulation Platforms

Our performance evaluation experiments consist of both full system simulations and direct machine measurements using hardware counters. Our hardware experiments consisted of running OLTP for 5000 transactions five times (after a warmup period), discarding the best and worst case numbers and averaging the remaining three. Using DCPI [1], we measured execution time, instruction cache misses, and instruction TLB performance. We used a couple of different Alpha multiprocessor platforms that are specifically mentioned in the results sections.

For our simulations, we use the SimOS-Alpha environment (our Alpha port of SimOS [24]) which was used in a previous study of commercial applications and has been validated against Alpha multiprocessor hardware [2]. SimOS-Alpha is a full system simulation environment that simulates the hardware components of Alpha-based multiprocessors (processors, MMU, caches, disks, console) in enough detail to run Alpha system software. The ability to simulate both user and system code under SimOS-Alpha is essential given the rich level of system interactions exhibited by commercial workloads. Our simulations run from a checkpoint that is taken when the workload is already in its steady state, and run for 500 transactions (after a warmup period) on a simulated 4-processor Alpha system. The basic SimOS-Alpha simulations use a 1 GHz single-issue pipelined processor model with 64KB 2-way instruction and data caches (64-byte line size), and a 1.5MB 6-way unified L2 cache. The memory latencies assume aggressive chip-level integration [4]: 12ns L2 hit, 80ns for local memory, and 150-200ns for 2-hop and 3-hop remote misses. To simplify our study of instruction cache behavior with various cache parameters, we modified SimOS-Alpha to also generate application and kernel instruc-

tion traces. These traces include the virtual and physical addresses along with the CPU and process that executed the instruction, and are used with simple instruction cache simulators for our studies of miss behavior across a large variety of cache parameters.

After code layout optimizations, the improved efficiency of the workload often causes it to be more I/O bound. This leads to higher idle time, making elapsed execution time comparisons meaningless. In practice, the workload can be re-tuned to eliminate the excess idle time. However, such re-tuning would further modify the behavior of the optimized runs with respect to the unoptimized runs, making it difficult to isolate the effects of our optimizations. Therefore, we chose to use non-idle execution cycles instead of elapsed execution time as our performance metric.

4 Behavior of the Database Application in Isolation

Code layout optimizations typically yield performance improvements primarily due to improvements in the instruction cache behavior of the application. To better understand these effects, this section analyzes the instruction cache behavior of the application in isolation, i.e., without considering the impact of operating system interference. We achieve this by filtering out operating system instructions from the instruction stream before doing our cache simulations. The combined instruction stream is studied in the next section.

This section begins by presenting a detailed characterization of the instruction cache behavior of both the original and optimized database application binaries. The optimized binary incorporates all the optimizations described in Section 2. Next we show how the individual optimizations contribute to the overall miss reductions. Finally, we provide further insight into the changes in the application behavior that lead to these improvements.

4.1 Instruction Cache Miss Characterization

Figure 3 shows an execution profile of the unoptimized database application. The x-axis corresponds to the static instruction footprint sorted from the most to the least frequently executed instruction. The y-axis corresponds to the cumulative fraction of executed instructions that a given footprint size captures. The figure shows that a 50KB footprint only captures 60% of the executed instructions, and capturing 99% of the instructions requires nearly 200KB. The total footprint is around 260KB. Due to non-ideal packing of instructions into cache lines, the actual footprint assuming 128-byte lines is approximately 500KB. For reference, the static size of the baseline application binary is over 27MB. The large instruction footprint and flat execution profile highlight the extreme memory system demands of OLTP workloads.

Figure 4 shows the application instruction cache miss behavior for the baseline and optimized binaries, for a range of cache and line sizes. Figure 5 is a composition of the results in Figures 4(a) and (b) to more clearly indicate the improvements in misses in the optimized binary relative to the baseline. Overall, a 128-byte cache line seems to be the sweet spot across various cache sizes for both the baseline and optimized binaries. Focusing on 64KB and 128KB instruction caches (realistic for near-term systems), the relative reduction of misses due to code layout optimizations is approximately 55%-65%.

Overall, relative gains are larger with larger line sizes. This is due to the better packing of instructions into cache lines (more on this later). Furthermore, the relative gains are larger with larger cache sizes up to 256KB. This result is slightly counter-intuitive:

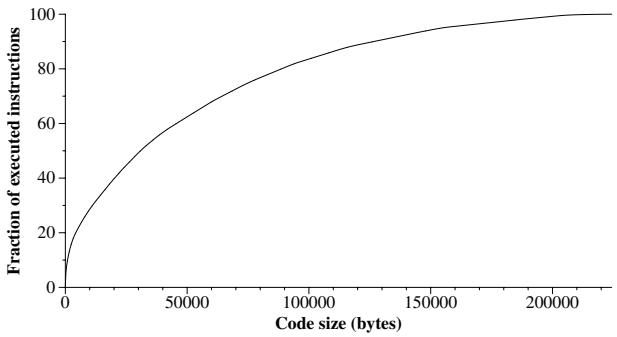


Figure 3: Execution profile of the unoptimized application binary: fraction of all dynamic instructions captured with a given footprint.

the better the baseline, the harder it should be to improve it. Yet, it appears that the code layout optimizations can make better use of the larger caches as well.

Figure 6 displays the impact of associativity for both the baseline and optimized binaries. For reasonable cache sizes (32KB-128KB), the impact of associativity is quite small. This is because capacity issues dominate at these sizes. In comparison, the benefits from code layout optimizations are much larger. These optimizations not only reduce conflicts by careful ordering of code segments, but also reduce capacity misses by better packing the code.

An interesting indication of how efficiently the code layout optimizations pack the application code comes from measuring the footprint of the baseline and optimized binaries in number of unique cache lines touched during execution. The optimized binary footprint in 128B cache lines is 37% smaller than the baseline binary (315KB versus 500KB). Furthermore, only 21% of the total number of instructions fetched into the cache are not used for the optimized binary, compared to 46% for the baseline binary (use/reuse of fetched instructions is discussed in more detail later in the section).

Figure 7 isolates the impact of different code layout optimizations on the instruction cache misses of the application. Procedure ordering alone causes a slight increase in cache misses. Clearly, the order of code segments at this large a granularity does not provide a better code layout. The largest (absolute) benefit comes from basic block chaining due to significant increases in instruction sequence lengths. Adding splitting or procedure ordering alone does not improve performance significantly. However, doing procedure ordering after the fine-grain routine splitting can further improve performance significantly; procedure ordering at this granularity separates frequently executed segments from infrequent ones, which further improves code packing.

4.2 Detailed Analysis of the Instruction Cache Miss Reductions

In this section, we use various metrics such as instruction sequence lengths, word usage before replacement, and cache line lifetimes, to provide further intuition.

The increased benefit of using larger cache lines with the optimized binary point to a significant increase in spatial locality. To explore this further, we measured the number of sequentially executed instructions between control breaks. Figure 8(a) shows the average number of sequentially executed instructions in both the baseline and the optimized application binaries. For reference, we also show the average dynamic basic block size which is com-

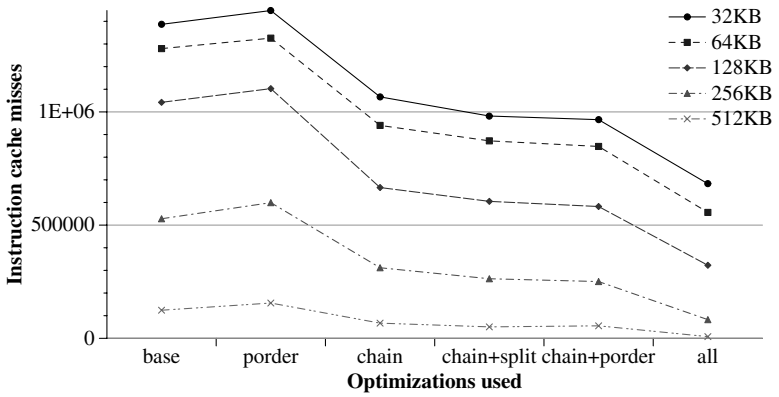
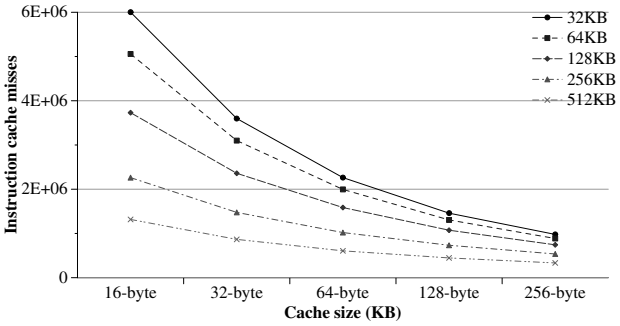
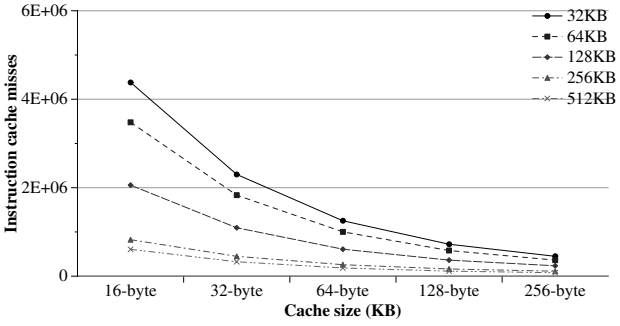


Figure 7: Impact of the different code layout optimizations on instruction cache misses.



(a) Baseline OLTP binary



(b) Optimized OLTP binary

Figure 4: Instruction cache misses for various cache and line sizes. Caches are direct-mapped; 4-processor system.

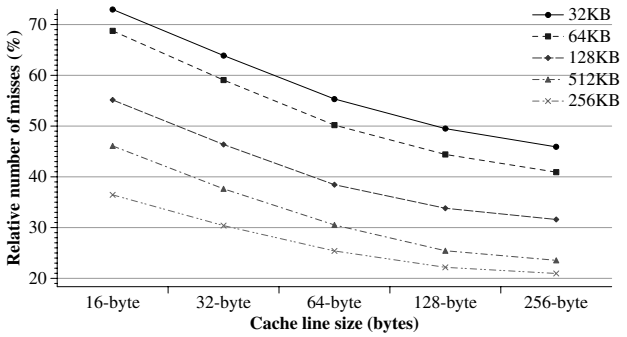


Figure 5: Relative instruction cache misses for the optimized binary over the baseline binary. Caches are direct-mapped; 4-processor system.

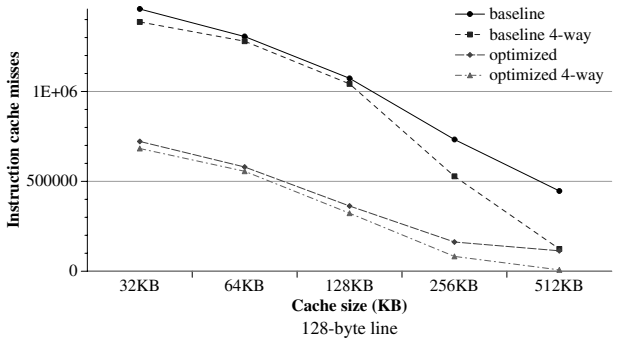
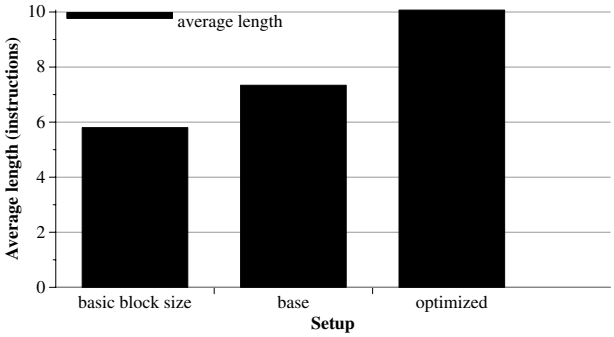
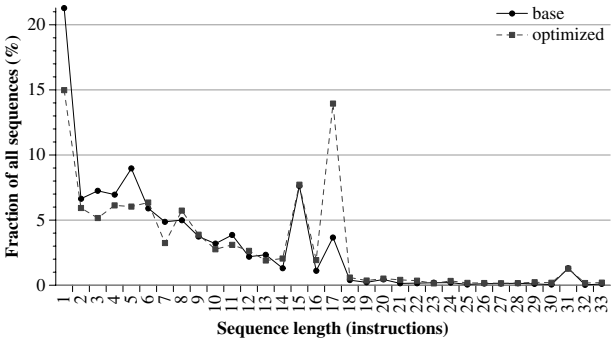


Figure 6: Impact of associativity on instruction cache misses for both the baseline and the optimized binaries.



(a) Average number of sequentially executed instructions. The average basic block size is presented for comparison purposes.



(b) Percentage of sequences of each length.

Figure 8: Sequentially executed instructions in both the baseline and optimized application binaries.

mon to both binaries. The basic block chaining optimization aligns branches so that they tend to be not taken, increasing code sequentiality. The optimized Oracle binary increases the sequence length from 7.3 to over 10 instructions. Figure 8(b) shows a histogram of the sequence length for both binaries. The results show that the sequence length increase in the optimized binary is mainly due to a reduction of the 1-instruction sequences (from 21% to 15% of all sequences), and a large spike at the 17-instruction length (14% of all sequences), which is just above the 64-byte cache line size. Even for the unoptimized binary, the large number of sequences at length 15 suggests that a larger line size than 64-bytes is beneficial (since sequences are not guaranteed to be aligned with a cache line).

While the increases in instruction sequence lengths are noticeable, they alone do not explain the instruction miss improvements shown before. Figure 9 shows a more direct measure of the increased spatial locality. The figure provides a histogram of the number of unique words used in a 128-byte cache line before it is replaced from a 128KB cache. The results show that the code layout optimizations lead to a remarkable increase in the number of times the full 128-byte cache line is used before being replaced (over 60% of the lines!).

We next consider a metric for temporal locality. Figure 10 shows the number of times an individual instruction is used before eviction. The results shows that over half of the instructions fetched into the cache are not used in the the unoptimized binary, and very few instructions are used more than once. In comparison, the optimized binary significantly reduces the number of unused

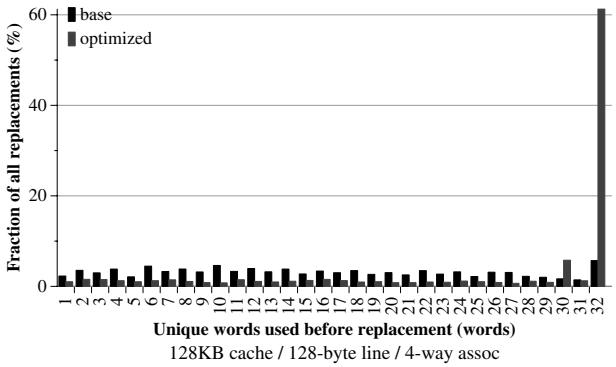


Figure 9: Unique word usage before cache replacement.

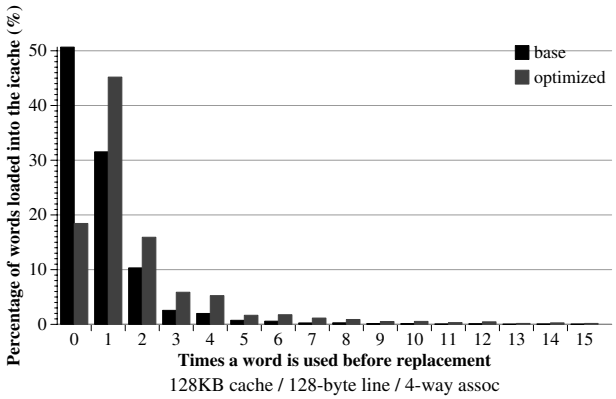


Figure 10: Histogram of individual instruction reuse before cache replacement.

fetched instructions and increases the number of instructions that are used more than once before eviction. Figure 11 provides another metric for temporal locality by showing cache line lifetime measured in cache cycles (i.e., number of cache accesses). On average, the lifetime increases by over a factor of 2 due to code layout optimizations.

In summary, code layout optimizations provide a major reduction in application instruction cache misses. The gains primarily come from longer instruction sequences and better packing of frequently executed sequences which lead to improved spatial and temporal locality.

5 Combined Database Application and Operating System Behavior

The previous section studied the application instruction stream in isolation to develop insight into the interaction between the code layout optimizations and the application behavior without involving potentially complex interactions between the application and operating system footprints. Previous studies have shown that OLTP applications exhibit significant operating system activity [2, 17]. The interactions between the application and operating system instruction streams are analyzed in this section.

Figure 12(a) shows the number of instruction cache misses for the combined instruction streams of the unoptimized application

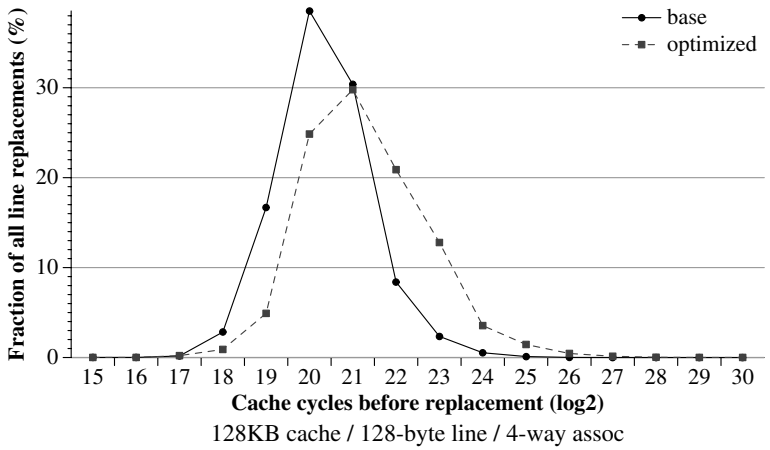


Figure 11: Cache line lifetimes for base and optimized binaries.

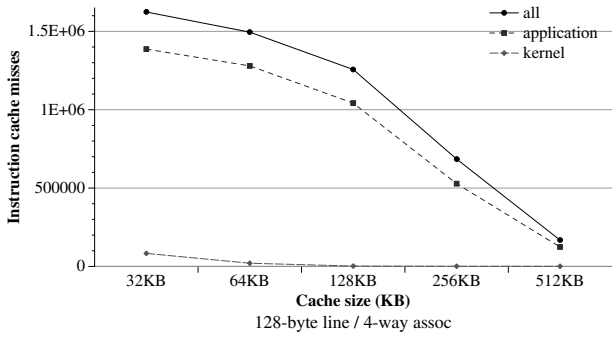
and the operating system. The two dotted curves show the number of misses if the instruction streams were executed in isolation. The solid curve shows the effect of the combined streams. Even though the number of misses in the kernel is small when executed in isolation, the impact of the interference between the kernel and application footprints causes a noticeable increase in total misses. Figure 12(b) shows the same data except with an optimized application binary. The effect of the interference with the kernel is more pronounced in this case primarily because instruction misses from the application are substantially reduced by the layout optimizations while the number of misses due to interference remain constant. Overall, the reduction in instruction misses is 45%-60% for 64-128KB caches. This compares to the 55%-65% reductions when we study the application in isolation.

Figure 13 shows a detailed analysis of the interference between the application and the operating system. For each instruction cache miss from a given process, the graph shows the owner process of the displaced cache line. For both the baseline and optimized application binaries, the majority of application misses arise due to self interference. In contrast, the kernel interferes very little with itself, with the majority of kernel misses caused by interference with the application.

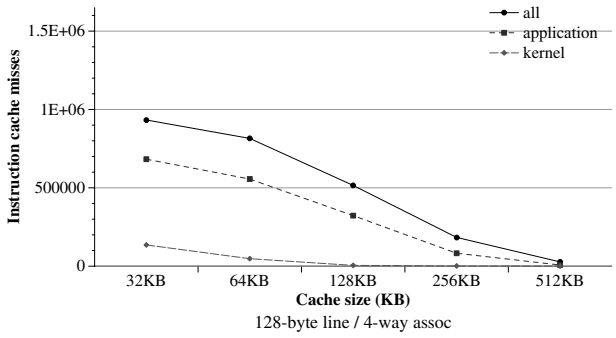
In addition to the direct impact on instruction cache misses, code layout optimizations can also improve behavior in other parts of the memory hierarchy. Figure 14 shows the number of misses in the instruction TLB and the shared L2 cache (with data and instruction misses shown separately) for both the baseline and the optimized Oracle binaries. These results are derived from our base SimOS simulations assuming a 64-entry fully associative iTLB and a 1.5MB 6-way set-associative L2. The decrease in iTLB misses primarily arises from better code packing at the page granularity. The reduction in L2 cache misses is less intuitive. These reductions also arise from the better code packing at the cache line granularity which leads to less interference with both other instruction and data cache lines.

We have also measured the impact of the code layout optimizations on instruction, iTLB misses, and board-level cache misses in a 21164-based AlphaServer. Our results indicated a 28% reduction in instruction misses (8KB cache), a 43% reduction in iTLB misses (48-entry iTLB), and a 39% reduction in board-level cache misses (2MB direct-mapped).

Finally, Figure 15 presents execution time measurements on two separate Alpha platforms: a 21264-based (AlphaServer DS20,

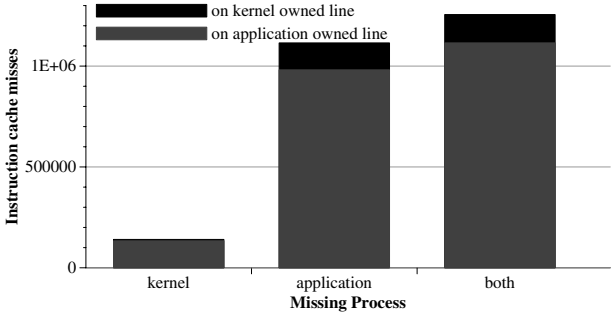


(a) baseline OLTP binary

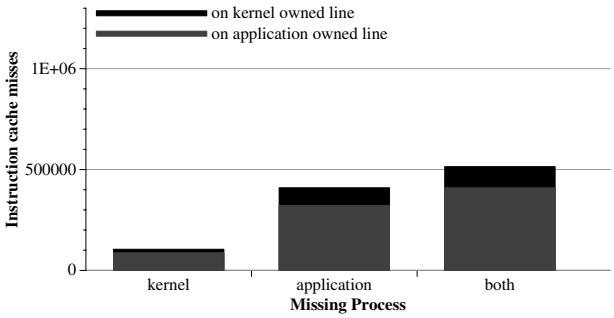


(b) optimized OLTP binary

Figure 12: Instruction cache behavior for combined application and operating system.



(a) baseline OLTP binary



(b) optimized OLTP binary

Figure 13: Interference between application and kernel instruction streams.

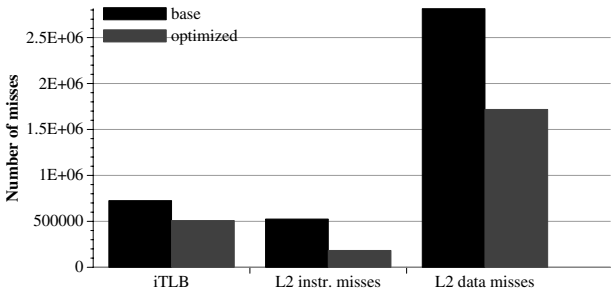


Figure 14: Comparison of the iTLB and L2 cache performance for the baseline and optimized binaries.

600MHz) and a 21164-based (AlphaServer 4100, 300MHz) server. The figure shows the reduction in execution time relative to the unoptimized (base) application binary for various combinations of layout optimizations. Overall, both systems achieve an improvement of 1.33 times in execution time due to the optimizations. Our SimOS simulations that approximate a 1GHz Alpha 21364-based system show a 1.37 times improvement in execution time. It is interesting to note that the overall execution time improvements are consistent across three generations of Alpha processors with substantially different clock frequencies and memory system parameters. The above results are for single-processor runs. Multiprocessor runs can reduce the impact of code layout optimizations due to the increased impact of data communication misses. For example, a 4-processor run on the AlphaServer 4100 yields a 1.25 times improvement in execution time (compared to the 1.33 times improvement for the 1-processor run). Finally, we also performed the same code layout optimizations on the operating system. However, the improvement in performance was much smaller (3.5%), partly because kernel execution constitutes a small fraction of the overall execution time. (A combined code layout optimization of the application and the kernel may provide more synergistic gains; however, we did not study this.) The above improvements from code layout optimizations with Spike are consistent with results we have observed in audit-sized TPC-C benchmarks on AlphaServers.

6 Discussion and Related Work

Code layout optimizations were originally proposed to reduce the working set size of applications for virtual memory [8, 10, 11]. More recent work has focused on the reduction of branch mispredicts and cache misses.

McFarling [18] describes an algorithm that uses the loop and call structure of a program to determine which parts of the program should overlay each other in the cache and which parts should be assigned to non-conflicting addresses.

Hwu and Chang [13] describe a profile based algorithm which uses function inline expansion, and groups basic blocks that tend to execute in sequence into traces. Traces that tend to execute close together are mapped in the same page to avoid conflicts among them.

Pettis and Hansen [20] propose a profile based technique which first reorders the basic blocks inside a procedure to reduce the number of taken branches. Then, procedures are split in two parts: the hot section which contains the frequently executed code, and the cold part which contains mostly unused code. After splitting, the procedures are mapped in memory so that two procedures which call each other will be mapped close in memory. As described before, we use a fine-grain procedure splitting technique instead of the hot/cold technique.

Torrellas *et al.* [28] designed a basic block reordering algorithm for operating system code. Using a basic block chaining algorithm similar to that of Hwu and Chang, they build traces spanning several functions, and then keep a section of the cache address space reserved for the most frequently referenced basic blocks.

Gloy *et al.* [9] extend the Pettis and Hansen placement algorithm at the procedure level to consider the temporal relationship between procedures in addition to the target cache information and the size of each procedure. Hashemi *et al.* [12] and Kalamaitianos *et al.* [14] use a cache line coloring algorithm inspired by the register allocation graph coloring technique to map procedures to minimize the number of conflicts. The above studies do not consider procedure splitting and/or chaining in combination with the procedure placement algorithm. Our work shows that procedure placement is quite ineffective on its own for workloads such as

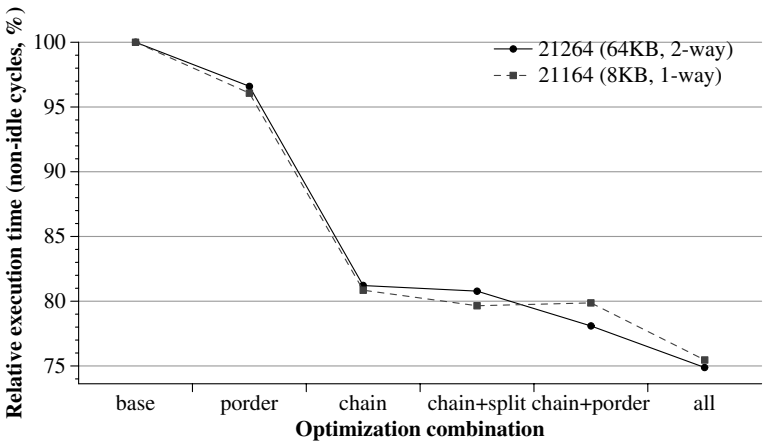


Figure 15: Execution time of various code layout optimizations on Alpha hardware platforms.

OLTP which have large instruction footprints, and only becomes beneficial when used in combination with splitting and chaining.

Ramirez *et al.* [21, 22] use a variation of the Torrellas algorithm specially targeted to database applications, and map whole basic block traces to the reserved area of the instruction cache in an effort to maximize code sequentiality. Their work was mainly on DSS which has a much better instruction cache behavior than OLTP.

Code layout has been implemented in various production tools, such as Compaq's Object Modification tool (OM) [27], its successor Spike [5], and IBM's FDPR [26]. They are all variations of Pettis and Hansen's algorithm.

Maynard [17] and more recently Barroso *et al.* [2] and Keeton *et al.* [15] have characterized the memory system behavior of commercial workloads. Our paper specifically examines how code layout changes this behavior. Ranganathan *et al.* [23] analyze the performance of commercial applications on out-of-order machines, and show that a 4-element instruction stream buffer proves effective at increasing performance. Their work is based on application traces without the kernel, and hence does not include the interference between application and kernel instruction streams. Nevertheless, our analysis of code layout optimizations suggests that it can be used to enhance the efficiency of instruction stream buffers by increasing instruction sequence lengths.

Cohn *et al.* [5] and Schmidt *et al.* [26] measure the effect of code layout on a wide variety of applications. Cohn *et al.* report that code layout makes most applications faster, except for small programs that spend most of the time in tight loops. Schmidt *et al.* only consider code layout optimizations for the operating system. In comparison, our paper contains an in-depth analysis of code layout optimizations in the context of OLTP workloads.

7 Concluding Remarks

With the growing dominance of commercial applications in the multiprocessor server market, it is important to consider software and compiler-level optimizations in addition to hardware features that can improve the performance of such workloads. This paper presents an in-depth analysis of utilizing profile-driven compiler optimizations to improve code layout in the context of OLTP workloads which are especially challenging due to their large instruction footprints.

Considering the application instruction stream in isolation, the code layout optimizations lead to a 55% to 65% reduction in the instruction misses for 64-128K caches, with basic block chaining being responsible for a large portion of this improvement. The improvement in instruction misses can be primarily attributed to longer sequences of consecutively executed instructions and better spatial and temporal reuse of cache lines before they are replaced. The combined application and operating system instruction streams highlight the importance of interference between the two streams, which becomes even more prominent with the optimized application. Applying code layout optimizations to the operating system code does not lead to any significant improvements. The overall impact of code layout optimizations on the combined instruction stream is a reduction of 45% to 60% in misses. We also show that optimizing the code layout can provide significant indirect improvements in other parts of the memory system such as the instruction TLB and the unified second level cache. Overall, these optimizations yield an improvement in performance of 1.33 times across our different simulation and hardware platform experiments.

The typical size and complexity of commercial workloads makes them less amenable to research on software and compiler-level optimizations. However, we find the significant performance improvements achieved by using relatively simple profile-driven code layout optimizations to be quite promising. We hope that these encouraging results pave the path for further exploration of software techniques to improve the behavior of commercial workloads and reduce the burden on hardware designs.

Acknowledgments

We would like to thank Jennifer Anderson for her early involvement in this work. We also thank the anonymous reviewers for their comments.

References

[1] J.-A. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the

cycles gone? In *Proceedings of the 16th International Symposium on Operating Systems Principles*, pages 1–14, Oct 1997.

[2] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 3–14, June 1998.

[3] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[4] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, January 2000.

[5] R. Cohn, D. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997. http://research.compaq.com/wrl/DECarchives/DTJ.

[6] Z. Cventanovic and D. Bhandarkar. Performance characterization of the Alpha 21164 microprocessor using TP and SPECworkloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 60–70, Apr 1994.

[7] Z. Cvetanovic and D. D. Donaldson. AlphaServer 4100 performance characterization. *Digital Technical Journal*, 8(4):3–20, 1996.

[8] D. Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, Nov. 1974.

[9] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. *Proceedings of the 30th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, Dec. 1997.

[10] D. J. Hartfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 2:169–192, 1971.

[11] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, 14(11):1640–1644, Nov. 1988.

[12] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 171–182, June 1997.

[13] W.-M. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 242–251, June 1989.

[14] J. Kalamaitianos and D. R. Kaeli. Temporal-based procedure reordering for improved instruction cache performance. *Proceedings of the 4th Intl. Conference on High Performance Computer Architecture*, Feb. 1998.

[15] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 15–26, June 1998.

[16] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 39–50, June 1998.

[17] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Oct 1994.

[18] S. McFarling. Program optimization for instruction caches. *Proceedings of the 3rd Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, Apr. 1989.

[19] S. E. Perl and R. L. Sites. Studies of Windows NT performance using dynamic execution traces. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 169–184, Oct. 1996.

[20] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 16–27, June 1990.

[21] A. Ramirez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Optimization of instruction fetch for decision support workloads. *Proceedings of the Intl. Conference on Parallel Processing*, pages 238–245, Sept. 1999.

[22] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *Proceedings of the 13th Intl. Conference on Supercomputing*, June 1999.

[23] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out of order processors. *Proceedings of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[24] M. Rosenblum, E. Bugnion, S. A. Herrod, and S. Devine. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[25] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 285–298, 1995.

[26] W. J. Schmidt, R. R. Roediger, C. S. Mestad, B. Mendelson, I. Shavit-Lottem, and V. Bortnikov-Sitnitsky. Profile-directed restructuring of operating system code. *IBM Systems Journal*, 37(2), 1998.

[27] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.

[28] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 360–369, Jan. 1995.

[29] http://www.tpc.org.

[30] Transaction Processing Performance Council. *TPC Benchmark B (Online Transaction Processing) Standard Specification*, 1990.