Software Trace Cache

Alex Ramirez, Josep L. Larriba-Pey, *Member*, *IEEE*, and Mateo Valero, *Fellow*, *IEEE*

Abstract—This paper explores the use of compiler optimizations which optimize the layout of instructions in memory. The target is to enable the code to make better use of the underlying hardware resources regardless of the specific details of the processor/architecture in order to increase fetch performance. The Software Trace Cache (STC) is a code layout algorithm with a broader target than previous layout optimizations. We target not only an improvement in the instruction cache hit rate, but also an increase in the effective fetch width of the fetch engine. The STC algorithm organizes basic blocks into chains trying to make sequentially executed basic blocks reside in consecutive memory positions, then maps the basic block chains in memory to minimize conflict misses in the important sections of the program. We evaluate and analyze in detail the impact of the STC, and code layout optimizations in general, on the three main aspects of fetch performance: the instruction cache hit rate, the effective fetch width, and the branch prediction accuracy. Our results show that layout optimized codes have some special characteristics that make them more amenable for high-performance instruction fetch: They have a very high rate of not-taken branches and execute long chains of sequential instructions; also, they make very effective use of instruction cache lines, mapping only useful instructions which will execute close in time, increasing both spatial and temporal locality.

Index Terms—Pipeline processors, instruction fetch, compiler optimizations, branch prediction, trace cache.

1 Introduction

Superscalar processors represent the major trend in high-performance processors in the last several years. These processors naturally evolve from pipelined architectures and try to obtain higher performance in two ways: First, by simultaneously executing several independent instructions in parallel; second, by increasing the clock rate to speed up instruction execution.

When designing a high-performance processor, it is important to keep all parts of the processor balanced, avoiding bottlenecks whenever possible. If we design a high-performance processor capable of executing five ALU operations at once, it is also important to ensure that we can feed the ALU stage and retire those instructions without stalling the pipeline. This means fetching and decoding at least five instructions per cycle, to keep the ALU stage busy, and writing results and graduating instructions at a fast enough rate.

But, the fetch stage does not behave like other pipeline stages in the sense that it cannot be widened by simply replicating it or adding more functional units. Furthermore, it has to follow the control path defined by branch instructions which have not been executed yet. The fetch stage quickly evolved to include branch prediction and used it to fetch instructions from speculative execution paths.

This ability to follow speculative paths independently of the execution stages leads to a decoupled view of the processor. The fetch engine reads instructions from memory and places them in an instruction buffer following a speculative path indicated by the branch prediction mechanism. Then, an execution engine reads instructions from the buffer and generates the required results, providing feedback to the fetch engine regarding the actual outcome of branch instructions.

An analysis of the decoupled view of a superscalar processor reveals that there are three main factors in fetch performance: 1) **memory latency**: how long it takes to read the instructions from memory, 2) **fetch width**: how many instructions we can transfer each cycle, and 3) **branch prediction accuracy**: how many transferred instructions belong to the wrong execution path.

The time it takes to load the required instructions from memory is computed, together with the time it takes to execute the instructions. If the memory latency is large, it can quickly become the major component in the execution time. The main approach to reducing the memory latency is the use of cache memories and prefetching schemes. Given the popularity of this approach, instead of measuring instruction memory latency, we will measure the instruction cache miss rate.

As we mentioned earlier, the fetch engine cannot be widened by simply replicating its functional units. Fetching more than one instruction per cycle requires a completely new fetch architecture, capable of selecting which instructions are to be fetched. This fetch architecture also determines how many instructions can be fetched simultaneously. The ability to fetch multiple instructions in a single cycle becomes a more important fetch performance factor as the issue width of the processor increases.

Finally, we must consider the presence of branch instructions which disrupt the flow of instructions through the pipeline. The problem arises when the outcome of a branch is not known until several cycles after it has been fetched, but we need to continue fetching instructions from a speculative path. By the time the branch has been resolved, several wrong path instructions may have entered the pipeline and may need to be squashed. The squashing

• *The authors are with the Universitat Politecnica de Catalunya, Jordi Girona 1-3, Module D6, 08034 Barcelona, Spain.*
E-mail: {aramirex, larri, mateo}@ac.upc.es.

Manuscript received 21 Feb. 2003; revised 8 Aug. 2003; accepted 19 Aug. 2003; published online 16 Nov. 2004.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 118324.

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

of wrong path instructions represents a wasted amount of fetch cycles and directly affects fetch performance. The frequency of this event mainly depends on the accuracy of the branch prediction mechanism.

Given the importance of fetch performance in superscalar processors, we target an increase in the rate at which useful instructions can be provided to the execution core. However, we consider approaching fetch performance from the software perspective. We consider the use of compiler optimizations to adapt the existing applications to the underlying fetch architecture. This software approach is attractive for two reasons: First, it has a null hardware cost, it does not require additional transistors, and does not require additional power; second, it provides performance improvements on already existing architectures, which makes them immediately applicable.

The rest of this paper is organized as follows: In Section 2, we discuss previous related work in the field of code layout optimizations and how the software trace cache improves on them. Section 3 describes the software trace cache algorithm in detail. In Section 4, we describe the different benchmarks used and our simulation setup. Section 5 presents an in-depth analysis of the performance impact of the software trace cache on the different factors of fetch performance and the overall impact on processor and system performance. Finally, in Section 6, we present our conclusions for this work.

2 Related Work

The mapping of instruction to memory is determined by the compiler. This mapping determines not only the code page where an instruction is found, but also the cache line (or which set in a set associative cache) it will map to. Furthermore, a branch will be taken or not taken depending on the placement of the successor basic blocks.

By mapping instructions in a different order, the compiler has a direct impact on the fetch engine performance. In this section, we provide a brief description of the different algorithms proposed to select where each instruction should be mapped.

We can divide code layout optimizations in three parts: the layout of the basic blocks inside a routine, the splitting of a procedure into several different routines or traces, and the layout of the resulting routines or traces in the address space. In this section, we will describe some algorithms for each of these optimizations and point out the benefits which can be obtained from them.

2.1 Basic Block Chaining

Basic block chaining organizes basic blocks into traces, mapping together those basic blocks which tend to execute in sequence. There have been several algorithms proposed to determine which basic blocks should build a trace [2], [7], [13], [20], [23], [24], [32].

The chaining algorithm used in [13], [23], [24], [32] is a greedy algorithm, which, given a *seed* or starting basic block, follows the most frequent path out of it as long as that basic block has an execution frequency larger than a given *ExecThreshold* and the transition has a probability higher than a given *BranchThreshold*. This implies visiting the routine called by the basic block or following the most frequent control flow out of the basic block. If the most likely path out of a basic block has already been visited, the next possible path is taken. If there are no possible paths out of a basic block, or the available paths do not pass the Exec and Branch thresholds, the algorithm stops and the next seed is selected.

A second alternative is the *bottom-up* algorithm proposed in [20] and used in [5], [18], [31]. The heaviest edge in the graph (the edge with the highest execution count) is selected and the two basic blocks are mapped together. The next heaviest edge is taken and processed in the same way, building basic block chains. After all basic blocks have been mapped to chains, the different chains are mapped in order so that conditional branches map to forward/usually not taken branches.

However, a control flow graph with weighted edges does not always lead to a basic block representing the most frequent path through a subroutine. The solution to this problem is path profiling [2]. A path profile counts how many times each path through a subroutine was followed, not simply how many times a branch was taken/not-taken. In this case, the correspondence between the profile data and the basic block chains which should be built is immediate.

Our basic block chaining algorithm derives from [32]. As we show in Section 3, we improve their chaining algorithm by automating some parts of the algorithm which required human intervention, like the seed selection, and selecting the Exec and Branch threshold values.

2.2 Subroutine Splitting

After a new ordering of the basic blocks has been established for a given procedure, the frequently executed basic blocks are mapped toward the top of the procedure, while infrequently used basic blocks will move toward the bottom of the procedure body. By splitting the different parts of the procedure, we can significantly reduce its size, obtaining a denser packing of the program.

We can distinguish two main ways of splitting a procedure body. A coarse-grain splitting would split the routine in two parts [5], [18], [20], [31]: one containing those basic blocks which were executed in the profile (the hot section) and another one containing those basic blocks which were never executed for the profiling input (the cold section).

A fine-grain splitting would split each basic block chain as a separate procedure [13], [21], [32]. The end of a chain can be identified by the presence of an unconditional control transfer because, after reordering, it is assumed that all conditional branches will be usually not-taken. Unused basic blocks would form a single chain and be kept together in a new procedure.

The procedures resulting from splitting do not adhere to the usual calling conventions, there is no defined entry or exit point, and they do not include register saving/restoring. This is done to avoid overhead associated with standard procedure control transfers.

As we show in [21], the benefits of the procedure splitting optimization do not lay within the splitting itself: It reflects on the improvements obtained with the procedure

placement optimizations. Mapping smaller procedures gives these optimizations a finer grain control on the mapping of instructions without undoing what the basic block chaining optimizations obtained.

2.3 Procedure Placement

Independently of the basic block chaining and procedure splitting optimizations, the order in which the different routines in a program are mapped has an important effect in the number of code pages used (and, thus, on the instruction TLB miss rate) and on the overlapping between the different procedures (and, thus, on the number of conflict misses).

The simplest procedure mapping algorithm is to map routines in popularity order: the heaviest routine first and then in decreasing execution weight order. This limits conflicts among two equally popular routines.

The mapping algorithm used in [5], [18], [20], [21], [31] is based on a call graph of the procedures with weighted edges, where the edge weight is the number of times each procedure call was executed. This algorithm can be extended to consider the temporal relationship between procedures and the target cache size information as described in [10]. Starting from the heaviest edge, the two connected nodes are mapped together and all incoming/outgoing edges are merged together. When two nodes containing multiple procedures should merge, the original (unmerged) graph is checked to see which way they should join: The two groups will merge at the strongest relationship point in the original graph, reversing the order of one or both of them if necessary. For example, if there are two groups, AB and CD, and the strongest relationship in the original graph is between A and C, the final ordering would be BACD.

In [11], [14], an optimized procedure layout is generated by performing a color mapping of procedures to cache lines, inspired by the register coloring technique, taking into consideration the cache size, the line size, the procedure size, and the call graph.

The mapping algorithm used in [32] follows a completely different approach, not based on the calling frequency of the generated procedures. After all basic blocks have been mapped to chains, the chains are ordered by popularity. The most popular chains are mapped to the beginning of the address space, while the least popular ones are mapped toward the end.

In addition to mapping equally popular chains next to each other, a fraction of the instruction cache is reserved for the most popular basic blocks by ensuring that no other code maps to that same range of cache addresses. These basic blocks are pulled out of whichever chain they mapped to and moved into this privileged cache space. This ensures that the most frequently used basic blocks will never miss in the cache. This reserved cache space is called the *Conflict Free Area* (CFA). The size of the CFA is determined experimentally.

We improve on this mapping algorithm by keeping all the basic blocks in a chain together [24]. That is, instead of mapping individual basic blocks into the CFA, we map the whole chain, which increases spatial locality and avoids taken branches. We also include a heuristic to determine the CFA size automatically.

3 The STC Algorithm

The Software Trace Cache (STC) layout algorithm is largely based on the work of Torrellas et al. [32], but includes multiple improvements. We no longer use the ExecThreshold and BranchThreshold parameters used in [23], [24], [32]. Instead, we build all our basic block traces in a single pass of the algorithm, without any user intervention to determine threshold values. We use an automatic process for selecting the starting point of our basic block traces. Finally, we map whole basic block traces into the Conflict Free area (CFA) instead of mapping individual basic blocks.

3.1 Seed Selection

Our algorithm is based on profile information. Running the training set on each benchmark, we obtain a directed graph of basic blocks with weighted edges.

The first step, before we can organize the basic block set into traces, is to select the *seeds* or starting points for those traces. In [32], the operating system code is studied in detail to find the most frequent entry points and a few subroutines are selected. In [23], we analyze the code of a relational database management system (DBMS) and select the entry points for the different query operations as seeds.

However, a detailed analysis of source code is not always feasible. For this reason, we have selected *all* subroutine entry points as seeds. We maintain the list of seeds ordered by basic block weight: from the most frequently executed seed to the least executed one. We explore each seed in turn, ignoring those seeds which have already been included in a previous trace.

This automatic selection of seeds is an important advantage of the STC over previous work in which the seed basic blocks where selected by the user based on a detailed analysis of the dynamic behavior of the application or the analysis of source code.

3.2 Trace Construction

From the selected seed, we proceed using a greedy algorithm which follows the most likely path out of a basic block, recording the path followed as the required trace. The algorithm follows paths regardless of them crossing the subroutine boundary, effectively building traces which cross multiple subroutines. The trace ends when all targets from a basic block have been visited or a subroutine return for the main procedure is encountered.

Loops are handled in the same way. The algorithm follows the most likely path though the loop body until the backward branch edge is found. That back-edge is recognized because it leads to an already visited basic block. The main target of the branch (the back edge) has already been visited and, so, the secondary target (the fall-through path) is chosen. Loops are not unrolled by the STC algorithm.

For example, following the graph in Fig. 1, the algorithm starts from seed A1. From basic block A1, the algorithm selects the most likely outgoing path, which leads to block A2. From basic block A2, the most likely outgoing path leads to an already explored seed C1. Discarded block B1 is already a seed and will be explored later. The trace starting from seed C1 and containing blocks C1 to C4 (excluding block C5) is

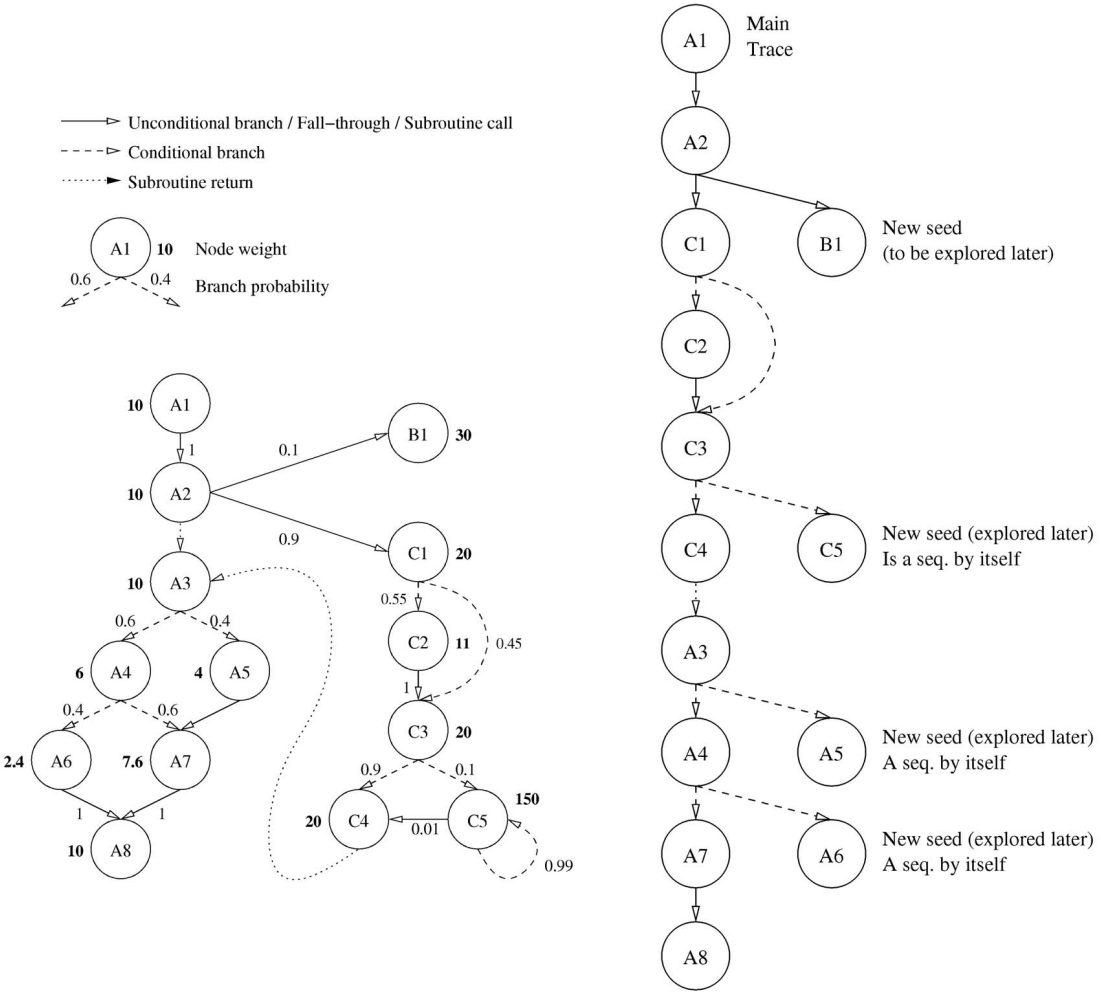

Fig. 1. Basic block chaining example.

then inlined after block A2. The algorithm continues at the next sequential block A3 (the return point for trace C1-C4).

From basic block A3, the most likely outgoing path leads to block A4. Discarded block A5 is added to the list of unvisited seeds, which is maintained in weight order. From basic block A4, the algorithm visits blocks A7 and A8, adding discarded block A6 to the seed list. Fig. 1 shows the resulting trace, including basic blocks from both routine A and routine C.

The chain inlining step is a novel contribution of the STC on top of what was done in [13], [32]. It allows the STC to build long basic block chains without the need for a careful seed selection based on source code analysis and makes the use of threshold values unnecessary.

3.3 Trace Mapping

As shown in Fig. 2, we map the resulting traces in the order they were created: from the most frequently executed one to the least executed one. In this way, we map equally popular traces next to each other, reducing conflicts among them. Also, we divide traces in instruction cache-sized chunks and leave an empty space at the beginning of each block except the first one (the one containing the most popular traces).

All code gaps map to the same place in the instruction cache so that there is no other code mapping to the same place as the most popular traces, creating a conflict-free area (CFA) for these traces which completely shields them from interference.

The size of the CFA is among the most determinant factors in the performance obtained using this mapping algorithm. A larger CFA fits more of the most popular traces, shielding them from interference, which reduces conflict misses in the most important segments of the code. However, it leaves less space in the instruction cache for the remaining traces, increasing conflict misses among them. Both factors balance each other and, after a given size, further increases in the CFA size actually decrease instruction cache performance.

As a difference with previous work, we use heuristics in order to determine an adequate CFA size without requiring a trial and error approach. We take the most popular traces, one at a time. Then, we compare the percentage of the total execution time that it gathers compared to the percent of the instruction cache that it requires. If the execution percent is higher than the space taken in the cache, we include the trace in the CFA. We then add the next trace and consider the percent of the execution they take together and the fraction of the cache they require. As long as the fraction of execution is larger than the fraction of the instruction cache they require,

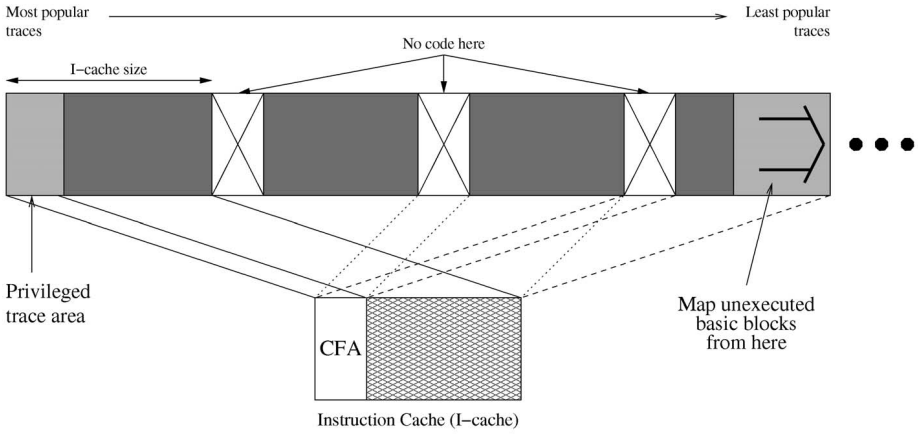

Fig. 2. Trace mapping example for a direct mapped instruction cache.

we keep adding traces to the CFA, trying to balance the two factors. For example, we would devote 32 percent of the instruction cache to the CFA when the traces contained gather 32 percent of the program execution.

This heuristic depends on the execution frequency of the traces built and the instruction cache size. For small caches, the size of the CFA will also be smaller, while larger caches allow for a larger CFA. Smaller codes which concentrate most of their execution in a few traces will almost completely fit in the CFA, while large codes with flat execution profiles will have little or no use for a CFA.

4 Simulation Setup

We have used a wide variety of workloads in this study, including typical integer applications (a subset of the SPEC'95 integer benchmarks)[1] [22], [24], a DSS workload (TPC-D on PostgreSQL) [23], [24], and an OLTP workload (TPC-B on Oracle v8) [21]. Table 1 shows which workload is used in each figure for the remainder of the paper.

For our DSS workload, we use a database with scale factor 0.1 (100MB of raw data) and run a representative subset of the read-only queries. We use a different subset of queries to obtain the profile data (Q3,4,5,6,9,15) and the performance simulation data (Q2,3,4,6,11,12,13,14,15,17).

The OLTP workload is set up and scaled in a similar way to what was done in [3]. We use a TPC-B database with 40 branches and a size of over 900MB. To hide I/O latencies, we use eight server processes per processor in this study.

Our performance evaluation and analysis experiments consist of a mix of detailed processor simulation, full system simulations, and direct machine measurements using hardware counters.

The processor simulator is derived from the SimpleScalar 3.0 tool set, extended with an aggressive fetch engine capable of fetching multiple sequential basic blocks in a single cycle (the SEQ.3 engine described in [27]), a trace cache [9], [19], [27], and dealiased branch predictors like agree [30], bimode [15], and gskew [17].

1. All except go for which we could not obtain profile information due to problems with the *pixie* tool and compress, which we considered too small to be representative.

For the full system simulations, we use the Alpha port of the SimOS environment [26]. SimOS-Alpha is a simulation environment which simulates Alpha multiprocessor hardware (processor, MMU, disk, caches) in enough detail to run system-level hardware and unmodified application code. Our simulations run from a checkpoint taken while the OLTP workload is in a steady state and run for 500 additional transactions on a simulated 4-processor Alpha system. Our SimOS setup uses a 1GHz single-issue pipelined processor with 64KB, 2-way instruction and data caches, and a 1.5MB unified L2 cache. Memory latencies assume chip-level integration: 12-cycles L2 hit, 80-cycles local memory, 120-150 cycles for 2-hop and 3-hop remote memory. We also use SimOS to obtain application traces which were used to analyze instruction cache behavior in detail.

The most significant results included in the paper are those obtained using real optimized applications on a real machine. The fact that the results obtained with this set up closely match those obtained via simulation makes us strongly confident of the validity of our study.

Our hardware experiments consisted of running our OLTP benchmark for 5000 transactions. Using DCPI [1], we measured execution time, instruction cache misses, and instruction TLB performance. We performed these experiments on two different Alpha platforms: an 8-processor 21164 and a dual processor 21264.

TABLE 1
Workloads Used in Each Section and Figure for This Paper

Sect.	Fig.	Workload
5.1	2.a	SPEC + TPC-D (PostgreSQL)
	2.b, 3, 4	TPC-B (Oracle DBMS)
5.2	5	SPEC + TPC-D
5.3	6, 7, 8	SPEC + TPC-D
5.4	9.a	TPC-B
	9.b	SPEC + TPC-D
	9.c	TPC-B

5 Performance Impact

This section presents our analysis of the impact of the STC and other code layout optimizations on all three aspects of fetch performance. Using detailed simulation of specific components and indirect performance metrics, we are able to explain the reasons for the performance improvements obtained.

Our results show that code layout optimizations not only improve instruction cache performance by avoiding conflict misses, but that they also make much better use of the available cache space, thus reducing *capacity* misses, and that spatial locality is the main advantage of optimized codes.

We also show that, after optimizations, it is possible to feed even the most aggressive superscalar processor by reading only chains of sequential instructions.

Our analysis of the impact of layout optimizations on the branch prediction mechanism shows that they can have a positive impact in the simple two-level adaptive predictors and a small negative impact on dealiased predictors. However, the improvements in other aspects of fetch performance overcome this slight drop in prediction accuracy.

Finally, we analyze the impact of layout optimizations in other elements beyond the fetch engine and find that they not only have a positive impact on the instruction memory hierarchy, but that they also improve *data* memory performance due to reduced interference between instructions and data.

5.1 Impact on the Instruction Cache

In this section, we examine the impact of code layout optimizations on the instruction memory latency, that is, how long it takes to fetch an instruction from memory. Because the main approach to reducing memory latency is the use of caches, the performance metric we use is the instruction cache miss rate.

Fig. 3a shows the the average instruction cache miss rate for the SPECint95 and the PostgreSQL database of a baseline cache setup (direct mapped) compared to that of the same cache running optimized codes, and two hardware optimized setups. The code layout optimizations explored are those proposed by Pettis and Hansen (PH) [20], Torrellas et al. (TXD) [32], and the Software Trace Cache (STC). The hardware optimized setups are a 2-way set associative cache and a 16-way fully associative victim buffer. None of the hardware optimized setups uses an optimized code layout.

The results in Fig. 3a show that code layout optimizations have a very significant impact on the instruction cache miss rate for all explored cache sizes, much larger than the two hardware optimizations explored. The instruction cache miss rate of a 16KB instruction cache running optimized codes is lower than that of a 64KB cache running unoptimized codes. This shows that optimized codes make more effective use of the available cache space, requiring a smaller cache to fit the instruction working set.

Comparing the STC with other code layout optimizations, our results show that the STC offers lower instruction cache miss rates than either the Pettis and Hansen or the Torrellas et al. optimizations, especially for the smaller cache sizes.

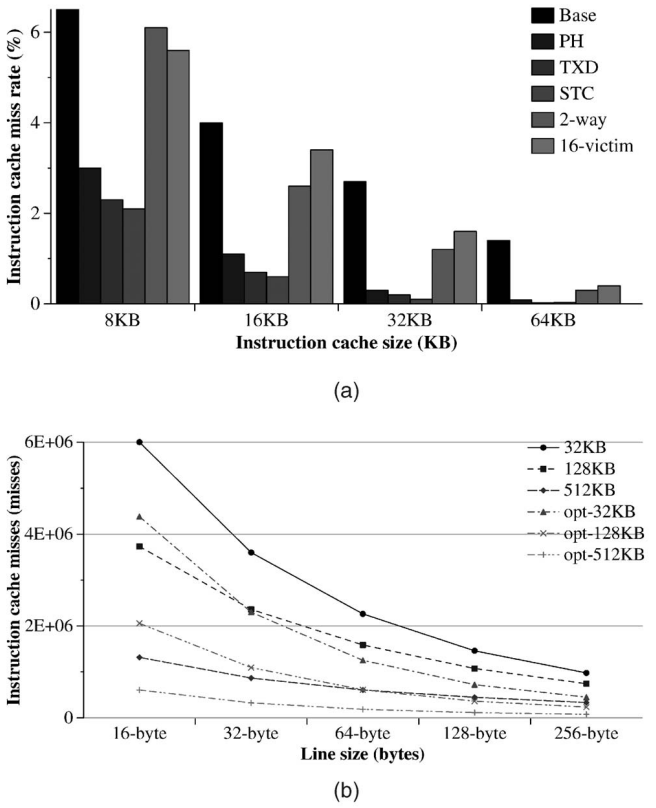

Fig. 3. Instruction cache misses for various cache configurations when using different hardware and code layout optimizations. (a) Miss rate comparison with other software and hardware optimizations. (b) Instruction cache misses on the DBMS application.

Code layout optimizations are very effective at reducing instruction cache miss rates. The usual explanation for this miss rate reduction is that a careful layout of the routines may reduce the number of conflict misses and that is the main aspect where code layout optimizations differ from each other. However, we will show that layout optimizations do not only have an impact on conflict misses.

Fig. 3b shows the number of instruction cache misses of two versions of a commercial database management system (DBMS) running an OLTP workload (TPC-B). Commercial databases are very large codes, with flat execution profiles, which suffer from heavy capacity problems rather than conflict misses.

The results in Fig. 3b show that code layout optimizations also have a significant impact on the number of misses of such big workloads, although the number of conflict misses cannot be reduced because the working set is too large to fit in the cache, regardless of the layout of routines.

It is also interesting to examine the relative number of misses of the optimized DBMS application compared to the unoptimized code, that is, for each instruction cache and line size, the percentage of misses still present in the optimized application. For example, on a 64KB cache with 128-byte lines, the optimized binary has only 45 percent of the misses of the unoptimized code.

The analysis shows that, even for large workloads which do not fit in the instruction cache, code layout optimizations can obtain important miss reductions (up to 76 percent reduction for a 512KB cache with 256-byte lines).

Further analysis of these results show that larger caches obtain better miss reductions. In our present case, this trend holds up to the 512KB cache because the workload already fits there, even in the unoptimized form. The same trend is present for the instruction cache line size: Longer cache lines obtain better miss reductions. The results in Fig. 3b show that the unoptimized application does improve performance as the cache line and size increase, but the optimized application improves faster than the baseline.

This trend shows that layout optimized codes exploit larger caches and longer cache lines better than unoptimized ones. Next, we analyze the reasons for these improvements in terms of spatial and temporal locality.

5.1.1 Spatial Locality

Code layout optimizations modify the basic block mapping to align branches toward their not taken direction, increasing the number of sequentially executed instructions. This increase in the sequence length translates immediately into an increase in spatial locality.

Our results on the commercial DBMS system show that there is a significant increase in the average sequence length from the baseline to the optimized application: from 7 instructions to 10 instructions. However, this increase is not enough to justify all the improvements seen in the instruction cache performance.

Fig. 4a shows a detailed breakdown of the number of sequences of each length for both binaries. The graph shows that there is a 30 percent decrease in the number of sequences of length 1 and a large increase in the number of sequences of length 17. That is, we are reducing the number of short sequences and increasing the number of long sequences. However, there is still more spatial locality than that explained by the basic block chaining optimization.

Fig. 4b shows the percentage of times that a number of unique words are used in a 128-byte cache line before it is replaced (32 instructions per cache line), for both the baseline and optimized application.

The results in Fig. 4b show that the optimized application uses the whole cache line over 60 percent of the time. That is, in most cases, all instructions in a cache line will be executed at least once before that cache line is replaced. Such behavior is not present in the unoptimized application and would explain the improved instruction cache performance.

The basic block chaining optimization alone does not explain this full usage of cache lines as most executed sequences are not long enough to fill an entire cache line. It is the combination of the routine splitting and the procedure ordering optimizations that causes this high percentage of cache lines to be fully used.

The routine splitting optimization separates the useful instructions from those which will rarely or never be executed, which reduces the size of the procedure. Then, the procedure ordering moves the useless instructions away and maps procedures which execute close in time next to each other. After this optimization, we not only execute

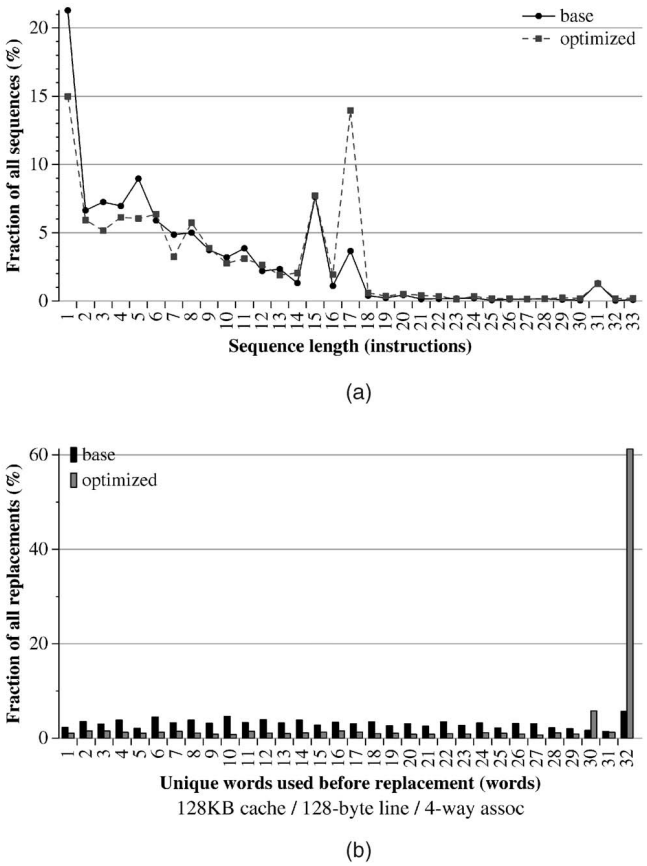

Fig. 4. Reasons for the spatial locality increase: increased code sequentiality and increased cache line usage in the commercial DBMS system. (a) Code layout optimizations increase the number of sequentially executed instructions. (b) Layout optimized codes use all the instructions in a cache line before it is replaced.

longer sequences of instructions, but, when a sequence is terminated, it is likely that the target sequence is in the same cache line.

By reducing the size of the procedures, optimized codes are able to better exploit larger sized caches by not wasting space to store instructions which will not be executed. And, they obtain higher improvements from longer cache lines because they exploit spatial locality, which increases significantly.

5.1.2 Temporal Locality

We have shown that optimized codes compact the useful sections of the code in a reduced number of cache lines, moving unused parts of the code toward the bottom of the program. This reduced size may have an impact on the temporal reuse of instructions.

Fig. 5 shows the number of cycles during which a given line has been present in the cache before being replaced. That is, we measure the lifetime of a cache line from the moment it is loaded into the cache to the moment it is evicted. Note that the X-axis showing the lifetime is in a logarithmic scale: A single step through the axis means the cache line was active for double the amount of time.

Our results show that cache lines have an extended lifetime in the optimized binary. The average lifetime has moved from 2^{19} cycles to 2^{20} or more cycles, meaning that

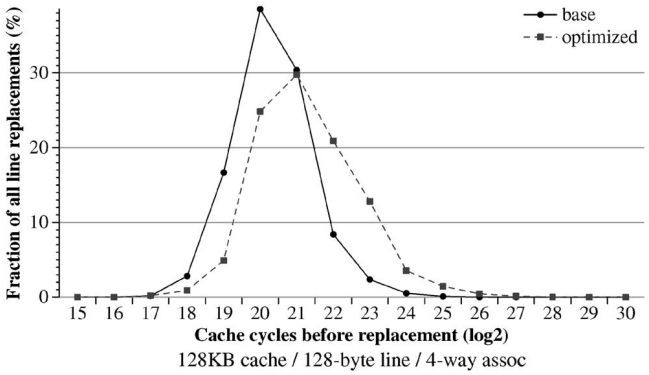

Fig. 5. Instruction cache lines have an increased lifetime in layout optimized codes.

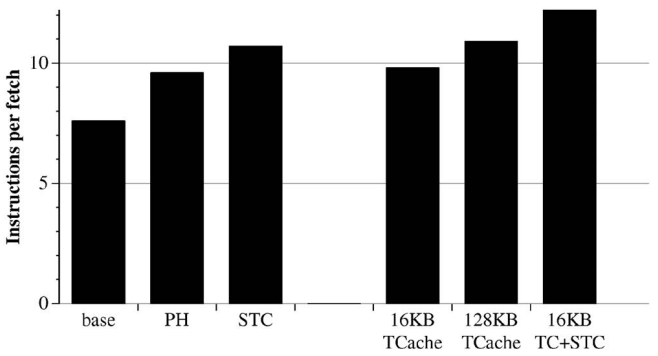

Fig. 6. Code layout optimizations effectively increase the fetch width of baseline and trace cache fetch architectures.

cache lines are available for twice the amount of cycles. Because we require fewer cache lines, we can keep a given cache line for longer before having to replace it, offering more opportunities for temporal reuse of instructions.

We have also measured the average number of times that a given instruction is used every time it is loaded into the cache. That is, every time we load a cache line, we count how may times each instruction was used before the line was replaced.

Our results show that the baseline (unoptimized) application does not use over 50 percent of what is loaded into the cache, while the optimized application uses over 80 percent of what is loaded (only 18 percent is left unused). This reflects the code compaction which we saw in the previous section.

If we examine the percentage of instructions which are used more than once, we see an increased reuse in the optimized application: 16 percent of all instructions are used twice, compared to a mere 10 percent in the unoptimized code. There is an increased percentage of instructions in all other reuse categories in the optimized application thanks to the increased lifetime of cache lines.

5.2 Impact on the Fetch Width

The layout of basic blocks in memory may also have an effect on the effective fetch width.

The presence of branches disrupts the fetch sequence, but it is taken branches which actually interrupt it. It is difficult to fetch both a taken branch and its target in the same cycle, as is done in the branch address cache [33] and the collapsing buffer [6]. It requires fetching multiple cache lines per cycle and a complex instruction alignment network, which may add extra pipeline stages.

Meanwhile, it is easy to fetch a not-taken branch and its target in the same cycle because they reside in consecutive memory positions. It is not necessary to fetch additional cache lines nor to realign the instructions to reflect the actual execution flow.

As will be shown in Section 5.3, code layout optimizations are very successful at aligning branches toward their not taken direction, reaching an 80 percent not-taken rate among conditional branches. Furthermore, 60 percent of all executed branches are always not taken.

Fig. 6 shows the impact of code layout optimizations on a fetch engine capable of fetching up to three sequential basic blocks per cycle (the SEQ.3 engine described in [27]) and a trace cache architecture.

The results in Fig. 6 show that code layout optimizations such as the one proposed by Pettis and Hansen [20] and the Software Trace Cache effectively increase the number of instructions provided by the fetch engine each cycle, reaching a performance level close to that of a trace cache.

Comparing the STC with the Pettis and Hansen optimized code, our results show that the STC offers a better fetch width, in addition to the improved instruction cache miss rate observed in the previous section.

But, the benefits of code layout optimizations are not restricted to architectures which fetch consecutive basic blocks. The trace cache allows the fetch engine to fetch nonconsecutive basic blocks in a single cycle, but it also experiences a significant performance boost when combined with code layout optimizations. Our results show that a small 16KB trace cache used on a layout optimized code has better performance than a much larger trace cache using unoptimized code.

The trace cache reads the dynamic instruction stream and, so, is unaffected by the layout of instructions in memory. However, the trace cache is not a standalone fetch mechanism. If the requested trace is not present in the trace cache, it has to be fetched from a secondary fetch path, usually a sequential fetch engine. It is in those cases when code layout optimizations help a small trace cache to increase performance: If the secondary fetch engine has a performance close to that of the trace cache, it is less critical to miss in the trace cache. A full comparison of the fetch performance of the STC and the trace cache can be found in [24], [25].

5.3 Impact on the Branch Predictor

We have shown that code layout optimizations have a positive impact on the instruction cache performance and that they increase the effective fetch width, but we have not examined the impact of the code layout on the branch prediction mechanism. In this section, we provide such an in-depth analysis, extending what was done in [4], [12].

A better instruction cache performance means that instructions can be provided faster, without waiting for the lower memory hierarchy levels. An increased fetch width means that, each time we fetch instructions, a larger amount

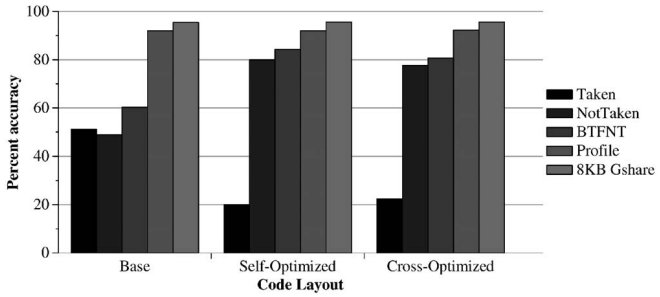

Fig. 7. Static branch prediction accuracy for the original and optimized code layouts (self and cross-trained).

of instructions is provided. But, if we negatively impact the branch prediction accuracy, we will be fetching very fast, and very wide, but from a wrong speculative path.

5.3.1 Effect on Static Prediction

In this section, we will examine the prediction accuracy that some simple static branch prediction schemes achieve for the examined benchmarks. The static strategies examined are: predict that all branches will be taken, predict that all branches will be not taken, predict that backward branches will be taken and forward branches will not, and predict that a branch will always take its most usual direction based on profile information [8], [29].

Fig. 7 shows the branch prediction accuracy of some simple static branch prediction strategies: always taken, always not taken, backward taken forward not taken (BTFNT), and the profile-based predictor, for both the original code layout and the compiler optimized layouts. For the optimized layout, we show results for the same input set used for training (self-optimized) and for a different input set (cross-optimized). The prediction accuracy of an 8KB gshare predictor [16] is shown for comparison purposes.

The simple static prediction approaches prove quite useless for the baseline code layout with near 50 percent prediction accuracy, only the BTFNT predictor reaches 60 percent and doesn't go under 50 percent for any of the studied benchmarks (individual benchmark results not shown). On the other hand, the profile static predictor proves very accurate, predicting correctly over 90 percent of the branches. This shows that branches can be predicted statically, but not with this simple strategy.

We optimize the code layout using the Software Trace Cache (STC) algorithm, which targets an increase in the sequentiality of the code, that is, it reorders basic blocks so that branches tend to be not taken.

Once we have optimized the code layout, the static branch prediction accuracy changes dramatically. The Not Taken and the BTFNT predictors now predict correctly over 80 percent of the branches, losing some accuracy in the cross-trained test. This 80 percent prediction accuracy shows that static branch prediction can be very accurate for these optimized code layouts, but it is still much lower than what can be achieved with modern two-level adaptive branch predictors like gshare.

To gain further insight on this high predictability of optimized binaries, we explore in depth the changes in branch behavior introduced by the code layout optimization.

Examining the branch classification for the original code layout, we observe that 36 percent of the branches are always not taken, while 32 percent are always taken. The rest of the branches are evenly spread across all taken percent values, with a slightly higher peak for branches that are 50 percent taken. This explains the low prediction accuracy obtained because branches do not seem to follow such simple behavior rules.

By optimizing the code layout, we can reverse the direction of those branches which are taken more than 50 percent of the time. This way, a branch which was taken 80 percent of the time will now only be taken 20 percent of the time.

The classification for the optimized code layout shows that we were quite successful at reversing the branch direction for those usually taken branches. The fraction of always taken branches is reduced from 32 percent to 10 percent and most categories over 50 percent taken also present reductions in the number of branches. This leads to a significant increase in the number of always not taken branches, from 36 percent to 59 percent. With most highly biased branches in the not taken side and most other branches moving from over 50 percent taken to mostly not taken, the prediction accuracy of an always not taken (or BTFNT) predictor increases significantly, as we have seen in Fig. 7.

Not all mostly taken branches can be reversed due to limitations in the algorithm. For example, loop terminations cannot be reversed (unless we perform loop unrolling)[2] and conditional branches explored late in the algorithm may find themselves with only one open path to follow, corresponding to the taken target. These explain why we could only reduce always taken branches from 32 percent to 10 percent.

The increase in the number of usually not taken branches explains the different behavior of the two code layouts regarding static branch prediction. Further increases in static prediction accuracy can be expected of a code layout optimization that explicitly targets a specific branch predictor, like the BTFNT predictor, or uses code replication techniques to use path information in its static predictions.

Next, we will examine how this change in branch direction affects dynamic branch prediction.

5.3.2 Effect on Two-Level Adaptive Predictors

Fig. 8a shows the effect of code reordering on dynamic prediction accuracy for the gshare [16], PAg [34], [35], and bimodal predictors [29]. Predictor sizes from 512 bytes to 16KB are explored for both the baseline (dotted line) and the optimized code layout (solid line).

Clearly, the STC increases the prediction accuracy of the examined branch predictors, especially for the smaller predictor sizes. Both the gshare and the bimodal predictors seem to converge at infinite predictor size, which points out that the benefits of using the STC are related to prediction

2. We considered a branch as always taken if it is taken over 95 percent of the time, so a loop with 20 iterations is terminated by an always taken branch.

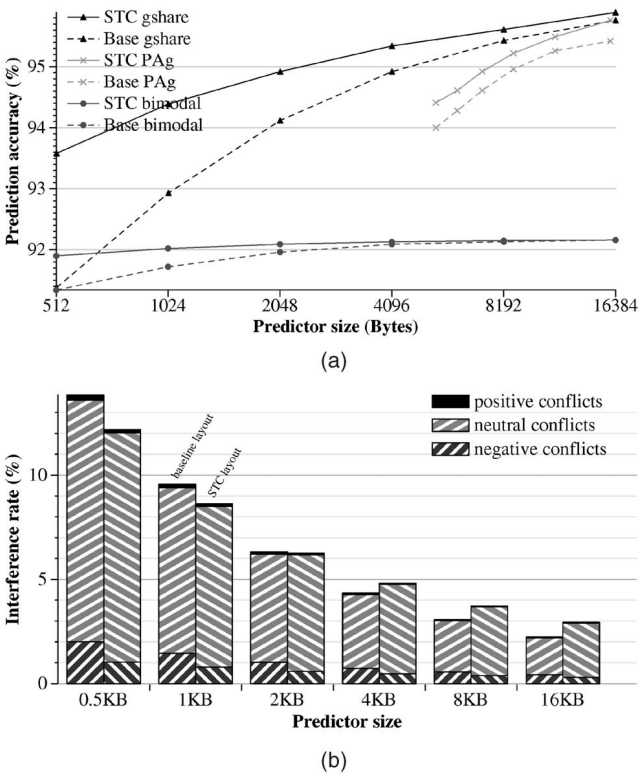

Fig. 8. Impact of code layout optimizations on two-level adaptive branch predictors. (a) Prediction accuracy using two-level predictors. (b) Dynamic branches which cause interference in the gshare prediction tables.

table interference. The larger the table, the less interference, the closer the prediction accuracy for both layouts.

Fig. 8b shows the percent of dynamic branches which introduce conflicts in the prediction tables of the gshare branch predictor with both the baseline and the optimized code layouts. We classify conflicts in three groups: neutral interference when the conflict does not change the prediction and positive or negative if the conflict changes the prediction for good or bad.

As expected, there is a significant reduction in the number of negative conflicts when the STC layout is used with the gshare branch predictor. For example, a 1KB gshare goes down from 1.45 percent of negative conflicts to 0.79 percent using the optimized code layout.

Intuitively, the increase in the number of not-taken branches favors positive interference because it is more likely that, when two branches interfere, they both behave the same way (both not taken), resulting in a positive or neutral conflict.

The total amount of conflicts shows a different behavior. The optimized code layout has fewer neutral conflicts for small predictor sizes, but it ends up with a larger amount of neutral interference for the largest configurations.

We will look further into this neutral interference increase in the next section, where we will examine dealiased branch prediction schemes.

5.3.3 Effect on Dealiased Predictors

Given that the use of an optimized code layout is reducing the negative interference found in the dynamic prediction tables, it is interesting to examine what happens with modern branch predictors that are already organized to minimize such interference, like the agree [30], bimode [15], and gskew [17], [28] predictors. We will refer to these predictors as dealiased branch prediction schemes.

Fig. 9 shows the prediction accuracy of the dealiased predictors with both the baseline and the optimized code layouts. The prediction accuracy of the gshare predictor with the optimized layout is shown for reference purposes.

These results show that, for small predictor sizes, the use of optimized code layouts obtains equivalent or higher accuracy, even in the dealiased branch predictors. The advantage of the optimized layouts is especially clear in the 0.4KB gskew predictor, which increases prediction accuracy from 93.5 percent to 94.4 percent.

For medium and large predictor sizes, all dealiased branch predictors obtain higher accuracy with the baseline code layout, the difference being especially significant with the 16KB agree predictor, which obtains a 96.2 percent accuracy with the baseline layout and a 95.8 percent with the optimized code.

A more important result shows that the use of a large agree or bimode predictor with the optimized code layout does not yield significant improvements over a gshare predictor. Only the gskew predictor obtains significantly better results than the gshare predictor when using the optimized code layout.

We have also examined the percent of dynamic branches which introduce conflicts in the prediction tables of the gshare branch predictor with the optimized code layout and the agree predictor using both code layouts.

These results show that the agree prediction scheme with a nonoptimized layout obtains a slightly better negative interference reduction than the optimized code layout. It is surprising that, using the agree predictor, the optimized code layout has more negative conflicts than the baseline.

From these results, it seems that the dealiased predictors prove more effective at reducing interference than the optimized code layout, but the more important result is that it seems more difficult to reduce conflicts in an optimized binary. The fact that the optimized code layout has more total interference for the larger predictor sizes can explain this higher fraction of negative conflicts.

The fact that a dealiased predictor like agree, using an optimized binary, obtains worse results than a gshare predictor points to some other factor hindering the performance of these predictors.

The high fraction of not taken branches found in the optimized code layout (80 percent of all branches are not taken) may be hindering the branch distribution in the BHR. When working with an optimized binary, the BHR will tend to be full of zeros, causing many possible BHR values to be never or rarely used, leading to a worse branch distribution and a loss of *useful* information to make a correct prediction.

We have analyzed this BHR distribution factor by counting the number of times each possible history value was found in an 11-bit global history predictor for both code layouts. The results show that the baseline code evenly spreads the usage of all possible BHR values, with a high peak at the value with all 1s (all taken branches), while the

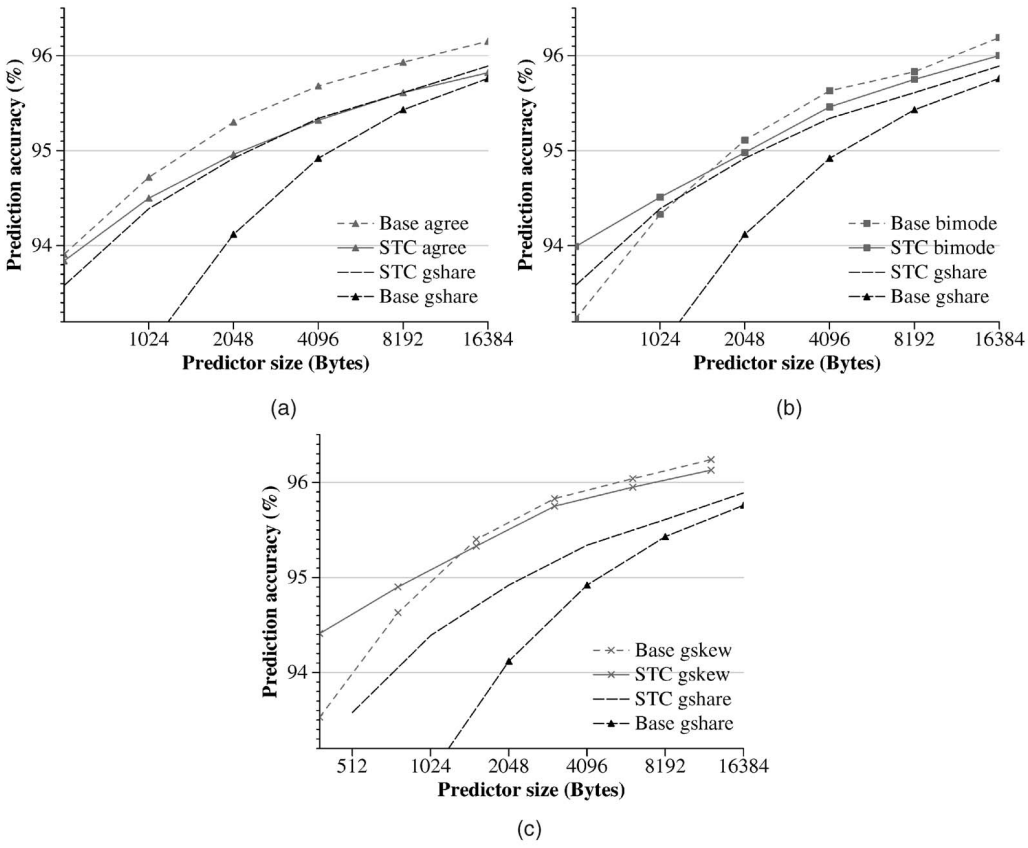

Fig. 9. Effect of the optimized code layout on dealiased branch predictors. (a) Agree predictor. (b) Bimode predictor. (c) Gskew predictor.

STC layout has a much higher concentration of uses in the values with many 0s (85 percent of all BHR values had eight or more 0s and 20 percent were all 0s).

This negative effect is especially visible in the GAg predictor, which depends exclusively on the correct distribution of the BHR values. The interference reduction cannot compensate for the poor value distribution, causing a loss of accuracy when using optimized codes.[3]

This does not happen in the gshare predictor because it XORs the branch address with the branch history, hiding this effect and causing the interference reduction effect to dominate.

The dealiased predictors do not benefit from the interference reduction effect because they are quite good at reducing it themselves, thus they only suffer the negative BHR effect and lose accuracy with the optimized code layout.

5.4 Overall Performance Impact

In this section, we examine the impact of code layout optimizations on the overall processor and system performance. Although code layout optimizations usually target the L1 instruction cache performance, they have a significant impact on other components of the fetch engine and other levels of the memory hierarchy.

Fig. 10a shows the number of misses in the instruction TLB and the shared L2 cache for a commercial database management system running an OLTP benchmark, using both unoptimized and optimized code. The misses in the shared L2 cache have been classified as either instruction misses, or data misses.

Our results show a reduction in the number of instruction TLB misses. Procedure placement optimizations move unused routines toward the end of the procedure, condensing the useful code in fewer pages, which explains this result.

The L2 shared cache shows a significant reduction in the number of instruction misses as a consequence of the careful layout of routines and basic blocks. A code which has been mapped to avoid conflicts in the L1 will also avoid conflicts in the larger L2.

A more surprising result is the significant reduction in L2 data misses. The increase in instruction spatial locality makes the code fit in fewer code pages and the decreased L1 and L2 instruction miss rate leaves more space in the shared L2 cache for the data to sit more comfortably, reducing conflicts among data and instructions, which leads to fewer data misses.

These results show that code layout optimizations have a positive impact not only on the L1 instruction cache, but on all levels of the memory hierarchy. This allows the performance improvements to go beyond what could be obtained by merely improving the instruction cache miss rate.

Fig. 10b shows the average processor performance measured in instructions per cycle (IPC) for the SPECint95 benchmarks using unoptimized and optimized codes for a variety of instruction cache sizes and a perfect instruction cache. Results are shown for a processor with a realistic branch predictor and a perfect branch predictor.

3. Results not shown for brevity, see [22] for a complete set of graphs.

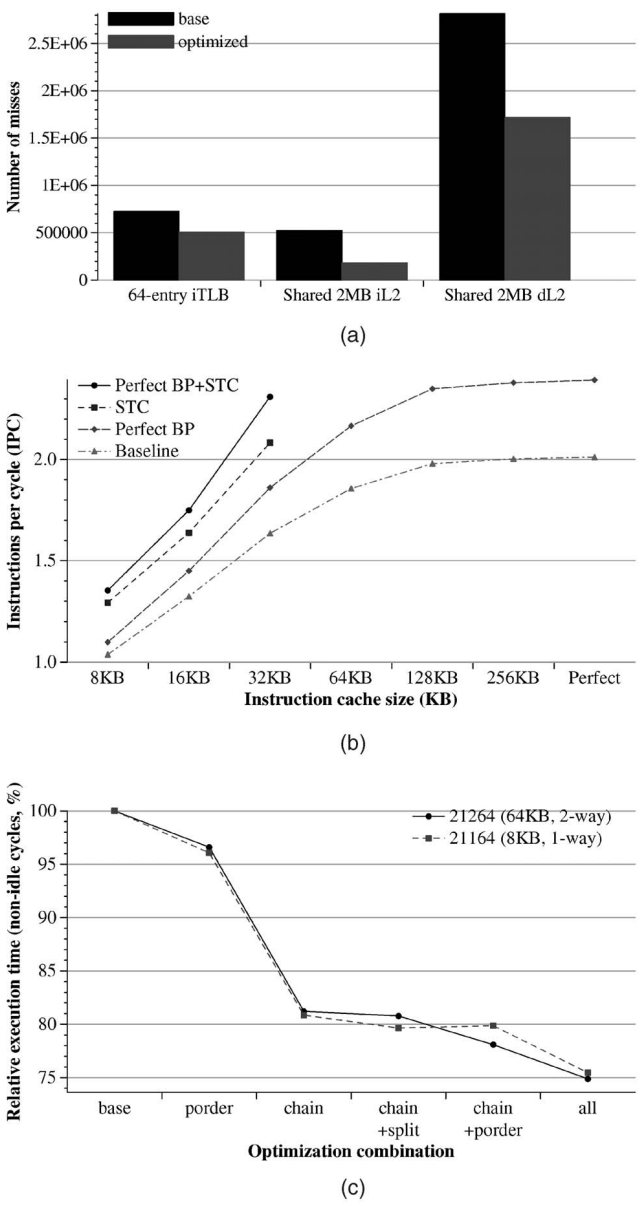

Fig. 10. Overall performance impact of code layout optimizations. (a) Impact on the memory hierarchy. (b) Impact on the processor IPC (SPEC-Int). (c) Impact on the overall system performance (OLTP).

The results in Fig. 10b show that processor performance using layout optimized codes is higher than that of unoptimized codes using an instruction cache of twice the size. Moreover, the performance of the unoptimized binaries saturates after 128KB are devoted to the instruction cache, while the performance of optimized codes with a 32KB cache is higher than that of unoptimized codes using a perfect instruction cache.

There is more than just an instruction cache performance improvement to consider: A fetch width increase, a better branch prediction accuracy, a lower TLB miss rate, and fewer data misses to the L2 all contribute to increasing performance.

When using perfect branch prediction, the improved prediction accuracy advantage of the optimized binaries dissolves and unoptimized codes can reach a higher performance. Still, optimized codes using a 32KB instruction cache reach the same performance as unoptimized codes on a 128KB cache.

Fig. 10c shows the relative execution time of our commercial database application as we include different code layout optimizations. The optimization combinations explored include: procedure ordering alone (porder), basic block chaining alone (chain), basic block chaining with procedure splitting (chain+split), basic block chaining with procedure ordering (chain+porder), and all optimizations together (chaining, splitting, and ordering). We show results for real machine runs on two different Alpha platforms.

Our results show that most of the performance improvement obtained derives from the basic block chaining optimization, which is mainly responsible for the increased spatial locality experienced. The next big step in performance is encountered when we add routine splitting and procedure ordering on top of the basic block chaining. The routine splitting provides an extra degree of freedom to the procedure ordering optimization, which now can move away the unused portions of a routine, compacting the code so that most cache lines contain only useful instructions.

Overall, our results show that code layout optimizations can reduce execution time by 25 percent in a difficult and important workload domain such as commercial databases. Furthermore, our results show that the performance improvements obtained are consistent across different processor generations.

6 Conclusions

In this paper, we have described the Software Trace Cache (STC), a code layout optimization which targets not only the instruction cache performance, but also the effective fetch width of the fetch engine.

We analyze the performance impact of the software trace cache and other code layout optimizations on all three aspects of fetch performance: the instruction cache miss rate, the effective fetch width, and the branch prediction accuracy.

Our results show that code layout optimizations provide significant improvements to the instruction cache performance, not limited to a conflict miss reduction. Optimized codes make much more effective use of the available cache space, packing only useful instructions in a cache line and moving unused sections of the code toward the end of the executable. This tight packing of instructions leads to a high increase in spatial locality and an increased lifetime of cache lines, which offers extended opportunities for temporal reuse.

We also show that layout optimizations can increase the effective fetch width of the front-end engine. A fetch engine capable of fetching multiple consecutive basic blocks increases performance to a level close to that of a trace cache and a small trace cache using optimized codes has a performance higher than that of a much larger trace cache running unoptimized applications.

Having a positive impact on the instruction cache and the fetch width may be worthless if we are decreasing the branch prediction accuracy. But, we show that such is not the case. Layout optimized codes are more amenable to

branch prediction using either static branch predictors or simple 2-level adaptive branch predictors. Only for dealiased branch predictors did we experience a slight performance drop in the branch predictor. However, the loss in prediction accuracy is more than compensated by the increased cache hit rate and fetch width.

Finally, we also examine the impact of code layout optimizations on the whole memory hierarchy and find that optimized codes have not only better instruction memory performance, but also better data memory performance due to the reduced conflict rate between data and instructions. Our results show that processor performance increases beyond what could be provided by a mere instruction cache performance increase, confirming that fetch width, branch prediction accuracy, and data memory performance are also important performance contributions by code layout optimizations. Our experiments with a commercial database application running an OLTP workload on real machine runs show that layout optimized codes can reduce execution time by 25 percent.

In this paper, we have advocated the use of compiler optimizations to increase fetch and processor performance, without the need for complex and expensive hardware modifications. We have improved on previous work on code layout optimizations with the STC and analyzed, in detail, the reasons for the increased fetch and processor performance. Our results show significant performance improvements by adapting the software to the characteristics of the underlying hardware.

ACKNOWLEDGMENTS

This work would not have been possible without the collaboration of Josep Torrellas from the University of Illinois at Urbana-Champaign, Luiz Barroso, Kourosh Gharachorloo, Robert Cohn, Geoffrey Lowney, and the whole Western Research Lab. team. This work was also supported by the Spanish Ministry of Science and Technology under contract TIC-2001-0995-C02-01, Generalitat de Catalunya under grant 1998FI-00306-APTIND, and CEPBA.

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?" Technical Report 1997-16, Compaq Systems Research Lab., July 1997.

[2] T. Ball and J.R. Larus, "Efficient Path Profiling," *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1996.

[3] L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *Proc. 16th Ann. Int'l Symp. Computer Architecture,* pp. 3-14, June 1998.

[4] B. Calder and D. Grunwald, "Reducing Branch Costs via Branch Alignment," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 242-251, Oct. 1994.

[5] R. Cohn, D. Goodwin, P.G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables," *USENIX,* pp. 17-23, Aug. 1997.

[6] T. Conte, K. Menezes, P. Mills, and B. Patell, "Optimization of Instruction Fetch Mechanism for High Issue Rates," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 333-344, June 1995.

[7] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers,* vol. 30, no. 7, pp. 478-490, July 1981.

[8] J.A. Fisher and S.M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proc. Fifth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 85-95, 1992.

[9] D.H. Friendly, S.J. Patel, and Y.N. Patt, "Alternative Fetch and Issue Techniques from the Trace Cache Mechanism," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* Dec. 1997.

[10] N. Gloy, T. Blackwell, M.D. Smith, and B. Calder, "Procedure Placement Using Temporal Ordering Information," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* pp. 303-313, Dec. 1997.

[11] A.H. Hashemi, D.R. Kaeli, and B. Calder, "Efficient Procedure Mapping Using Cache Line Coloring," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 171-182, June 1997.

[12] D.L. Howard and M.H. Lipasti, "The Effect of Program Optimization on Trace Cache Performance," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 256-261, Oct. 1999.

[13] W.-M. Hwu and P.P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. 16th Ann. Int'l Symp. Computer Architecture,* pp. 242-251, June 1989.

[14] J. Kalamatianos and D.R. Kaeli, "Temporal-Based Procedure Reordering for Improved Instruction Cache Performance," *Proc. Fourth Int'l Conf. High Performance Computer Architecture,* Feb. 1998.

[15] C.-C. Lee, I-C.K. Chen, and T.N. Mudge, "The Bi-Mode Branch Predictor," *Proc. 30th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* pp. 4-13, Dec. 1997.

[16] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Compaq Western Research Lab., June 1993.

[17] P. Michaud, A. Seznec, and R. Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 292-303, 1997.

[18] R. Muth, "Alto: A Platform for Object Code Modification," PhD dissertation, Univ. of Arizona, Aug. 1999.

[19] S.J. Patel, D.H. Friendly, and Y.N. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism," Technical Report CSE-TR-335-97, Univ. of Michigan, May 1997.

[20] K. Pettis and R.C. Hansen, "Profile Guided Code Positioning," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 16-27, June 1990.

[21] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J.L. Larriba-Pey, G. Lawney, and M. Valero, "Code Layout Optimizations for Transaction Processing Workloads," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* July 2001.

[22] A. Ramirez, J.L. Larriba-Pey, and M. Valero, "The Effect of Code Reordering on Branch Prediction," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 189-198, Oct. 2000.

[23] A. Ramirez, J.L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero, "Optimization of Instruction Fetch for Decision Support Workloads," *Proc. Int'l Conf. Parallel Processing,* pp. 238-245, Sept. 1999.

[24] A. Ramirez, J.L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software Trace Cache," *Proc. 13th Int'l Conf. Supercomputing,* June 1999.

[25] A. Ramirez, O.J. Santana, J.L. Larriba-Pey, and M. Valero, "Fetching Instruction Streams," *Proc. 35th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* 2002.

[26] M. Rosenblum, E. Bugnion, S.A. Herrod, and S. Devine, "Using the Simos Machine Simulator to Study Complex Computer Systems," *ACM Trans. Modeling and Computer Simulation,* vol. 7, no. 1, pp. 78-103, Jan. 1997.

[27] E. Rotenberg, S. Benett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Ann. ACM/IEEE Int'l Symp. Microarchitecture,* pp. 24-34, Dec. 1996.

[28] A. Seznec and P. Michaud, "D-Aliased Hybrid Branch Predictors," Technical Report PI-1229, IRISA, Feb. 1999.

[29] J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Ann. Int'l Symp. Computer Architecture,* pp. 135-148, 1981.

[30] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 284-291, 1997.

[31] A. Srivastava and D.W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time," *J. Programming Languages,* vol. 1, no. 1, pp. 1-18, Dec. 1992.

[32] J. Torrellas, C. Xia, and R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," *Proc. First Int'l Conf. High Performance Computer Architecture,* pp. 360-369, Jan. 1995.
[33] T.-Y. Yeh, D.T. Marr, and Y.N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proc. Seventh Int'l Conf. Supercomputing,* pp. 67-76, July 1993.
[34] T.-Y. Yeh and Y.N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Ann. Int'l Symp. Computer Architecture,* pp. 124-134, 1992.
[35] T.-Y. Yeh and Y.N. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* pp. 257-266, 1993.

Alex Ramirez received the computer science degree in 1997 and the PhD degree in computer science in 2002 from the Polytechnic University of Catalonia (UPC). His research areas of special interest are profile-guided compiler optimizations, code layout optimizations, performance studies of user and system code-like database applications, and the design and implementation of the fetch stage of superscalar and multithreaded processors. He has been a student intern at Compaq's Western Research Lab. (Palo Alto, California) and Intel's Microprocessor Research Lab. (Santa Clara, California). Since 2000, he has been lecturing on operating systems and operating systems administration as an assistant professor. Currently, he is involved in research and development projects with Intel and IBM.

Josep L. Larriba-Pey received the computer science degree from the Polytechnic University of Catalonia (UPC) in 1989 and the PhD degree from the same university in 1996. He is an associate professor in the Computer Architecture Department at UPC. His current research interests are in the relation between the architecture of the computer, the compiler and the high level applications, with special interest in databases, the tuning of the basic sequential and parallel DBMS and Record Linkage operations, and, the design, analysis and tuning of sequential and parallel nonnumeric algorithms. At present, he is also involved in research and development projects with IBM and Institut Catala d'Oncologia related to DBMS performance and record linkage, respectively. He is a member of the IEEE.

Mateo Valero received the telecommunication engineering degree from the Polytechnic University of Madrid in 1974 and the PhD degree from the Polytechnic University of Catalonia (UPC) in 1980. He is a professor in the Computer Architecture Department at UPC. His current research interests are in the field of high-performance architectures. He has published approximately 250 papers on these topics. He served as the general chair for several conferences, including PACT-01, ISCA-98, and ICS-95, and has been an associate editor for the *IEEE Transactions on Parallel and Distributed Systems* for three years. He has been honored with several awards, including the Narcis Monturiol, presented by the Catalan Government, the Salvà i Campillo presented by the Telecommunications Engineer Association and the ACM, the King Jaime I by the Generalitat Valenciana, and the Spanish national award "Julio Rey Pastor" for his research on IT technologies. Since 1994, he has been a member of the Royal Spanish Engineering Academy. In 2001, he was appointed a fellow of the IEEE and he has been a fellow of the ACM since 2003.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.