Software Trace Cache for Commercial Applications *

Alex Ramirez, Josep Ll. Larriba-Pey,
Carlos Navarro, Mateo Valero
Universitat Politecnica de Catalunya
Jordi Girona 1–3, D6
08034 Barcelona (Spain)
{aramirez,larri,cnavarro,mateo}@ac.upc.es

Josep Torrellas

Department of Computer Science
University of Illinois
torrella@cs.uiuc.edu

December 9, 1999

Abstract

In this paper we address the important problem of instruction fetch for future wide issue superscalar processors. Our approach focuses on understanding the interaction between software and hardware techniques targeting an increase in the instruction fetch bandwidth. That is the objective, for instance, of the Hardware Trace Cache (HTC).

We design a profile based code reordering technique which targets a maximization of the sequentiality of instructions, while still trying to minimize instruction cache misses. We call our software approach, *Software Trace Cache (STC)*.

We evaluate our software approach, and then compare it with the HTC and the combination of both techniques. Our results on PostgreSQL show that for large codes with few loops and deterministic execution sequences the STC offers better results than a HTC. Also, both the software and hardware approaches combine well to obtain improved results.

1 Introduction

Future wide-issue superscalars are expected to demand a high instruction bandwidth to satisfy their execution requirements. This will put pressure on the fetch unit and has raised concerns that instruction fetch bandwidth may be a major limiting factor to the performance of aggressive processors. Consequently, it is crucial to develop techniques to increase the number of useful instructions per cycle provided to the processor.

The number of useful instructions per cycle provided by the fetch unit is broadly determined by three factors: the branch prediction accuracy, the cache hit rate and the number of instructions provided by the fetch unit per access. Clearly, many things can go wrong. Branch mispredictions cause the fetch engine to provide wrong-path instructions to the processor. Instruction cache misses stall the fetch engine, interrupting the supply of instructions to the processor. Finally, the execution of non-contiguous basic blocks prevents the fetch unit from providing a full width of instructions.

Much work has been done in the past to address these problems. Branch effects have been addressed with techniques to improve the branch prediction accuracy [12] and to predict multiple branches per cycle [23, 26]. Instruction cache misses have been addressed with software and hardware techniques. Software solutions include code reordering based on procedure placement [8, 7] or basic block mapping, either procedure oriented [18] or using a global scope [11, 24]. Hardware solutions include set associative caches, hardware prefetching, victim caches and other classic techniques. Finally, the number of instructions provided by the fetch unit each cycle can also be improved with software or hardware techniques. Software solutions include trace scheduling [5], and

*This research has been supported by CICYT grant TIC-0511-98 (UPC authors), the Generalitat de Catalunya grants ACI 97-26 (Josep L. Larriba-Pey and Josep Torrellas) and 1998FI-00306-APTIND (Alex Ramírez), the Commission for Cultural, Educational and Scientific Exchange between the United States of America and Spain (Josep L. Larriba-Pey, Josep Torrellas and Mateo Valero), NSF grant MIP-9619351 (Josep Torrellas) and CEPBA. The authors want to thank Xavi Serrano for all his help setting up and analyzing PostgreSQL.

superblock scheduling [10]. Hardware solutions include branch address caches [26], collapsing buffers [4] and trace caches [6, 22].

While all these techniques have vastly improved the performance of superscalar I-fetch units, they have been largely focused and evaluated on engineering workloads. Unfortunately, there is growing evidence that popular commercial workloads provide a more challenging environment to aggressive instruction fetching.

Indeed, recent studies of database workload performance on current processors have given useful insight [1, 14, 16, 17, 21, 25]. These studies show that commercial workloads do not behave like other scientific and engineering codes. They execute fewer loops and have many procedure calls. This leads to large instruction footprints. The analysis, however, is not detailed enough to understand how to optimize them for improved I-fetch engine performance.

The work in this paper focuses on this issue. We proceed in three steps. First, we characterize the locality patterns of a database kernel code and find frequently executed paths. The database kernel used is PostgreSQL [15]. Our data shows that there is significant locality and that the execution patterns are quite deterministic and highly predictable.

Second, we use this information to propose an algorithm to reorder the layout of the basic blocks in the database kernel for improved I-fetch. Finally, we evaluate our scheme via simulations. Our results show an instruction cache miss reduction of 70–90% for realistic instruction cache sizes and a 50% increase of the number of instructions executed between taken branches from 9 to 14. As a consequence, a 16 instruction wide sequential fetch unit using realistic branch prediction increases the fetch bandwidth from 4.4 to 5.6 instructions per cycle when using our proposed code layout.

The software scheme that we propose combines well with hardware schemes like a Trace Cache. The fetch bandwidth for a 16KB trace cache improves from 5.1 to 6 when combined with our software approach.

1.1 The Fetch Unit

The importance of instruction fetch is obvious since it is not possible to execute instructions faster than they can be fetched. But its importance is not limited to that. As processors become more and more aggressive, larger instruction windows will be included to detect *Instruction Level Parallelism* (ILP) among distant instructions in those program segments with a high degree of data dependency. Maintaining such a large instruction window requires a high performance fetch mechanism. Fetch speed becomes specially relevant at program startup and miss-speculation points where the instruction window is emptied and must be filled again.

Also, wider instruction issue, value prediction, instruction reuse, speculative memory disambiguation and other aggressive speculative techniques allow the execution of more instructions per cycle. To execute more than 5–6 instructions per cycle, fetching instructions from multiple basic blocks per cycle becomes necessary.

As shown in Figure 1, a natural extension of a present superscalar fetch unit allows fetching of multiple consecutive basic blocks per cycle. Increasing the branch predictor throughput to obtain multiple branch predictions per cycle allows the address and mask logic to obtain instructions from multiple consecutive basic blocks. Also, fetching multiple consecutive instruction cache lines, we can obtain basic block sequences which cross the cache line boundary.

In this case, the core fetch unit is limited to fetch consecutive basic blocks because the design does not allow it to predict the branch target address and fetch it in the same cycle. Instruction fetch proceeds from the same instruction cache line as long as the branch is predicted not taken. If it is predicted taken, the target address is predicted and fetching proceeds the next cycle from the predicted address.

The performance of this core fetch unit is determined by three factors: branch mispredictions, instruction cache misses, and the execution of non-consecutive basic blocks.

The trace cache mechanism allows fetching of non-consecutive basic blocks coming from different instruction cache lines. As shown in Figure 1, the fill unit captures the dynamic instruction stream and stores the built instruction sequences and the branch outcomes which lead to them in a special purpose cache. If the same starting instruction and branch outcomes are found again in the future, the whole instruction trace can be fetched from the trace cache without additional processing.

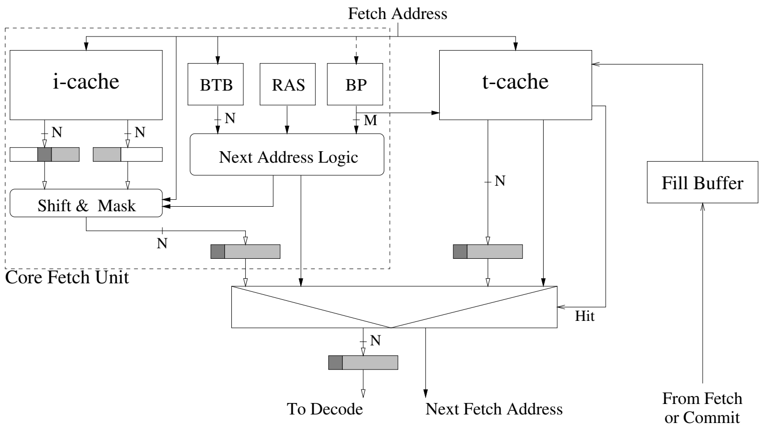

Figure 1: Core fetch unit able to fetch instructions from multiple consecutive basic blocks in a single cycle. Extension of the core fetch unit with a trace cache to allow fetching of non-consecutive basic blocks.

1.2 Related work

There has been much work on code mapping algorithms to optimize the instruction cache miss rate. These works were targeted at less aggressive processors, which do not need to fetch instructions from multiple basic blocks per cycle.

Hwu and Chang [11] use function inline expansion, and group into traces those basic blocks which tend to execute in sequence as observed on a profile of the code. Then, they map these traces in the cache so that the functions which are executed close to each other are placed in the same page.

Pettis & Hansen [18] propose a profile based technique to reorder the procedures in a program, and the basic blocks within each procedure. Their aim is to minimize the conflicts between the most frequently used functions, placing functions which reference each other close in memory. They also reorder the basic blocks in a procedure, moving unused basic blocks to the bottom of the function code, even splitting the procedures in two, and moving away the unused basic blocks.

Torrellas *et al* [24] designed a basic block reordering algorithm for Operating System code, running on a very conservative vector processor. They map the code in the form of sequences of basic blocks spanning several functions, and keep a section of the cache address space reserved for the most frequently referenced basic blocks. A comparison between the STC, the Pettis & Hansen method and the Torrellas *et al* method can be found in [19].

Gloy *et al.* [7] extend the Pettis & 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* [8] and Kalamaitianos *et al* [13] use a cache line coloring algorithm inspired in the register coloring technique to map procedures so that the resulting number of conflicts is minimized.

Techniques developed for VLIW processors, like Trace Scheduling [5] also identify the most frequent execution paths in a program. But these techniques are trying to optimize the scheduling of instructions in the execution core of the processor, not the performance of the instruction fetch engine. Individual instructions are moved up and down, crossing the basic block boundary, to optimize ILP in the execution core of the processor, inserting *compensation code* to undo what wrongly placed instructions did when the wrong path is taken. The traces they define are logical, the basic blocks need not be actually moved in order to obtain the desired effect. In that sense, these techniques and the STC may be complementary, one optimizes instruction fetch while the other optimizes instruction scheduling, both using the same profile information.

On the hardware side, techniques like the Branch Address Cache [26], the Collapsing Buffer [4] and the Trace Cache [6, 22] approach the problem of fetching multiple, non-contiguous basic blocks each cycle. The Branch Address Cache and the Collapsing Buffer access non-consecutive cache lines from an interleaved i-cache each cycle and then merge the required instructions from each accessed line. The Trace Cache does not require fetching of non-consecutive basic blocks from the i-cache as it stores dynamically constructed sequences of basic blocks in a special purpose cache. These techniques require hardware extensions of the fetch unit, and do not target an i-cache miss rate reduction, relying on other techniques for it.

Some other works have examined the interaction of run-time and compile-time techniques regarding the instruction fetch mechanism. Chen *et al.* [3] examined the effect of the code expanding optimizations (loop unrolling and function inlining) on the instruction cache miss rate. Also, Howard and Lipasti [9] examine the effect that different compiler optimizations like function inlining, loop unrolling and profile feedback compilation, have on the trace cache performance.

1.3 Paper structure

This paper is structured as follows. Section 2 analyzes the instruction reference stream for the PostgreSQL database system, examining both temporal and spatial locality issues. In Section 3 we propose the Software Trace Cache, a code reordering method which tries to exploit the exposed temporal and spatial locality. Section 4 presents the performance results obtained applying the STC to PostgreSQL, both regarding the instruction cache miss rate and the global fetch unit performance using a variety of instruction cache and trace cache sizes. Finally, Section 5 presents our conclusions for this work.

2 Workload analysis

The objective of the software trace cache (STC) is to build at compile time the most popular traces that are built at run time by the hardware trace cache. Also, the STC targets a minimization of the i-cache miss rate. We analyze the instruction reference stream for a relational database management system (PostgreSQL 6.3.2 [15]), characterizing instruction locality and execution path determinism, which affect the performance offered by the STC.

With this information we intend to predict the performance increase we can expect when using the STC on the database. The reference locality will affect the i-cache miss rate reduction offered by our technique, and the basic block size and the determinism of program execution will influence the increase in code sequentiality accomplished by the basic block reordering.

2.1 Temporal locality

Figure 2 shows the percentage of the dynamic instructions that we can gather with the most popular static instructions in PostgreSQL. The X axis shows code size instead of the plain number of instructions (each instruction is 4 bytes long).

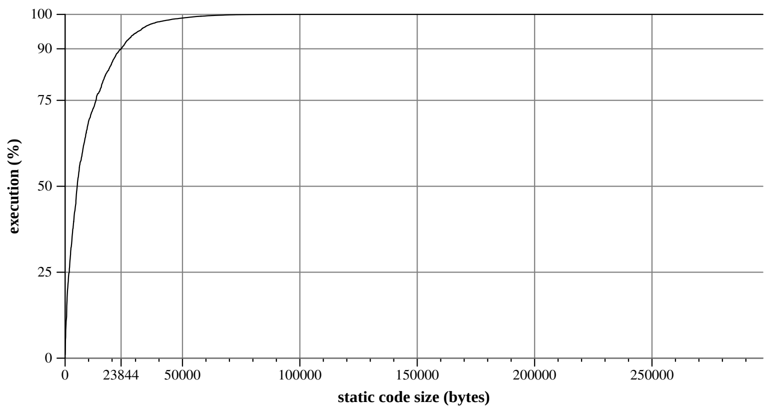

Figure 2: Percentage of the total number of instructions executed for static code size. A small number of static instructions gather most of the dynamically executed instructions.

It is clear that the static instructions are not equally popular. A small part of the code gather most of the dynamic instructions. With little more than 23KB of code we concentrate 90% of the executed instructions. With 32KB we gather 95% of the execution, and 64KB reach over 99%.

Meanwhile, the total instruction footprint for PostgreSQL is over 290KB, and the total code size is over 2MB. There are large sections of code which are never, or seldom, executed. This is code that must be present in the database code to cover special situations and rarely used data types.

From these results we conclude that a 64KB instruction cache can hold the entire instruction working set of PostgreSQL, if it is managed correctly. It would be interesting if we could ensure that these popular instructions which gather most of the execution would be always present in the instruction cache. As we explain in Section 3.2 this is exactly what we do.

2.2 Spatial locality

We begin our study of spatial locality by counting the number of sequential instructions executed by PostgreSQL. That is, the number of instructions between two taken branches/jumps.

Figure 3 shows the number of sequences of a given length that PostgreSQL executes. The average sequence length and the average basic block size are shown with a vertical line.

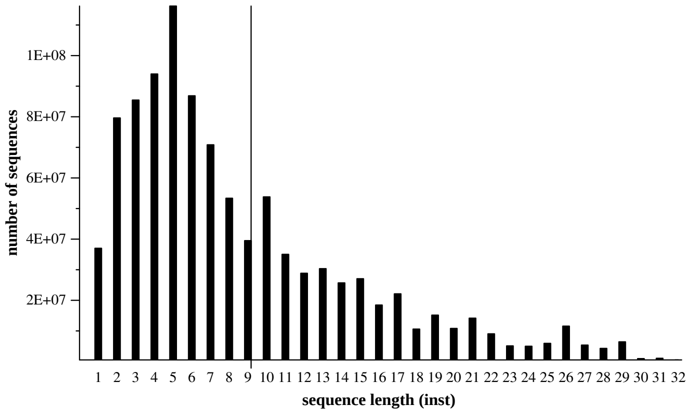

Figure 3: Number of times PostgreSQL executes a number of consecutive instructions between two taken branches/jumps. The average sequence length and the average basic block size are shown with a vertical bar.

The basic block size poses an absolute limit to the fetch bandwidth for any fetch unit that generates a single branch prediction per cycles. Such fetch units can not obtain more than 5–6 instructions per cycle for PostgreSQL. For a more aggressive fetch model with multiple predictions per cycle as the one shown in Figure 1, the limit is posed by the number of instructions executed sequentially. This means that we can rise the fetch limit to well over 20 instructions for PostgreSQL, with an average of 9–10 instructions, nearly twice as much as single basic block fetching.

The main problem of fetching instruction sequences is sequence fragmentation. While the average sequence is 9.2 instructions long, most sequences have lengths between 2–10 instructions. Longer sequences are rare, but it is possible to find a few very long ones. Once we pose a hardware limit to the sequence length, we can drastically reduce the fetch bandwidth. For example, if the hardware limit is 16 instructions, all sequences of length 17 would be broken in two, one of length 16 and one of length 1, for an average of 8.5 instructions.

It is possible to enlarge these sequences by aligning branches in a way that makes the not-taken target the most frequent one. Figure 4 classifies branches by the percentage of times they follow the taken path.

There is a large fraction of branches (92%) which always behave in the same way, always taken (45%), or always not taken (47%). But there is still 8% of branches which show a variable behavior. This shows that the execution sequence in PostgreSQL is quite deterministic.

It is possible to change a branch direction by doing some simple code transformations. Applying these code transformations to those usually-taken branches (50% taken or more), we can increase significantly the number of not taken branches, which will increase the sequence length and the fetch performance limit.

We must bear in mind that subroutine calls and returns, as well as indirect unconditional jumps, always break the execution sequence and cause a sequence termination. This limits the sequence length increase that we can obtain by performing the branch inversion.

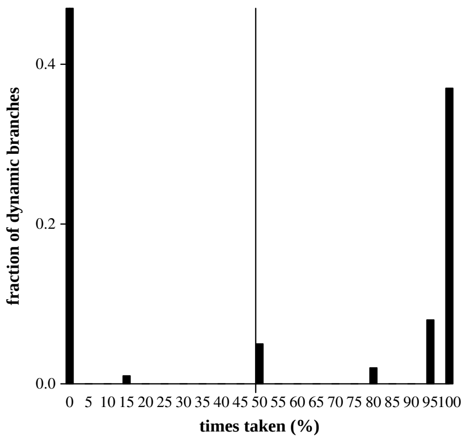

Figure 4: Percentage of branches classified by the number of times they follow the taken path. Average percent of taken branches shown with a vertical line (49%)

3 Code reordering

The code reordering algorithm is based on profile information. This means that the results obtained will depend on the representativity of the training inputs. The most popular execution paths for a given input set do not need to be related to the execution paths of a different input set.

Appendix A shows the criteria we use to select a representative subset of TPC-D on PostgreSQL. We use this subset of queries to obtain a directed graph of basic blocks with weighted edges. An edge connects two basic blocks p and q, if q is executed after p. The weight of an edge $W(pq)$ is equal to the total number of times q has been executed after p. The weight of a basic block $W(p)$ can be obtained by adding the weight of all outgoing edges. The branch probability of an edge $B(pq)$ is obtained as $W(pq)/W(p)$. All unexecuted basic blocks are pruned from the graph.

3.1 Building traces

In order to maximize the number of sequential instructions executed, we will arrange the basic blocks in a routine so that conditional branches tend to be not taken.

Using the weighted graph obtained running the training set, and starting from the routine entry points, we implement a greedy algorithm to build our basic block traces. Given a basic block, the algorithm follows the most frequently executed path out of it. This implies visiting a subroutine called by the basic block, or following the control transfer with the highest probability of being used. All the other valid transitions from the basic block are noted for future examination.

For this algorithm we use two parameters called *Exec Threshold*, and *Branch Threshold*. The trace building algorithm stops when all the successor basic blocks have been visited or have a weight lower than the Exec Threshold, or all the outgoing arcs have a branch probability less than the Branch Threshold. In that case, we start again from the next acceptable transition, as we noted before, building secondary execution paths for the same routine. Once all basic blocks reachable from the routine entry point have been included in the main or secondary sequences, we proceed to the next routine.

Figure 5.a shows an example of the weighted graph and Figure 5.b shows the resulting sequences. We use an ExecThresh of 4 and a BranchThresh of 0.4. Starting from node $A1$ and following the most likely outgoing edge from each basic block we build the sequence $A1 \rightarrow A8$ (Figure 5.b). The transitions to $B1$ and $C5$ are discarded due to the Branch Threshold. We noted that the transition from $A3$ to $A5$ is a valid transition, so we start a secondary trace with $A5$, but all its successors have been already visited, so the sequence ends there. We do not start a secondary trace from $A6$ because it has a weight lower than the Exec Threshold.

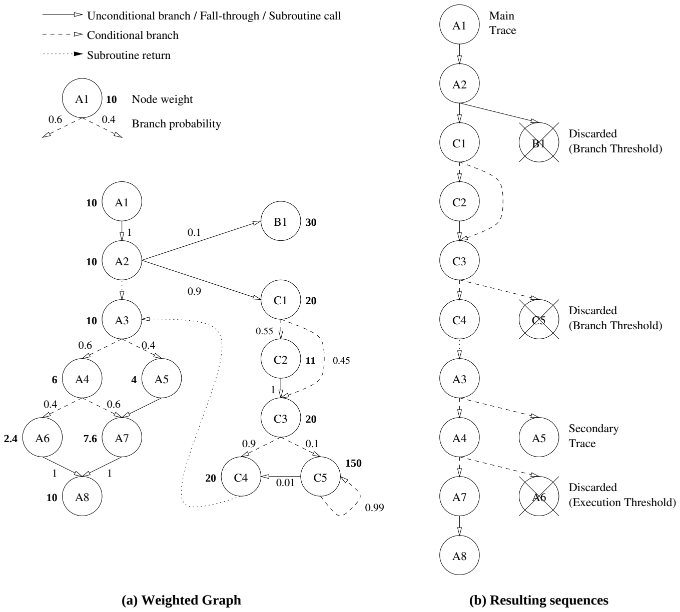

Figure 5: Example of the sequence building algorithm. The algorithm follows the most likely path out of a basic block, if that path passes a certain threshold. After the main path is finished, secondary paths are built.

3.2 Mapping traces

As shown in Figure 6, we map our code sequences in decreasing order of popularity, concentrating the most likely used code in the first memory pages and mapping popular sequences close to other equally popular ones, reducing conflict misses among them. Also, the most popular sequences will map to a reserved area of the cache, leaving gaps to create a Conflict Free Area (CFA), shielding the most popular traces from interference with any other code.

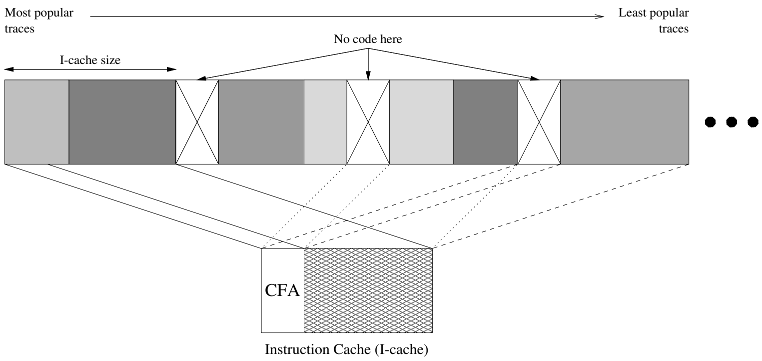

Figure 6: Sequence mapping for a direct mapped instruction cache.

Figure 7 shows how to apply the same mapping algorithm to set associative caches. The CFA size is split among the different sets, and the most popular sequences are mapped there. Next, the rest of the code is mapped across the remaining cache space, leaving the appropriate gaps to shield the routines in the CFA from interference.

The CFA sizing is heuristic. It follows a simple rule: the percentage of program execution gathered by the routines in the CFA must be larger than the percentage of the instruction cache that they use. To avoid degenerate cases, the CFA never uses more than 75% of the instruction cache.

Figure 8 illustrates this heuristic for an 8KB instruction cache. It is a trimmed version of Figure 2, where we look at the percentage of instruction cache space used instead of at the raw code size. The most popular code, filling 33% of the instruction cache is expected to gather 33% of the dynamically executed instructions. If we assign 50% of the instruction cache to the CFA, it will only gather 42% of the total executed instructions, wasting

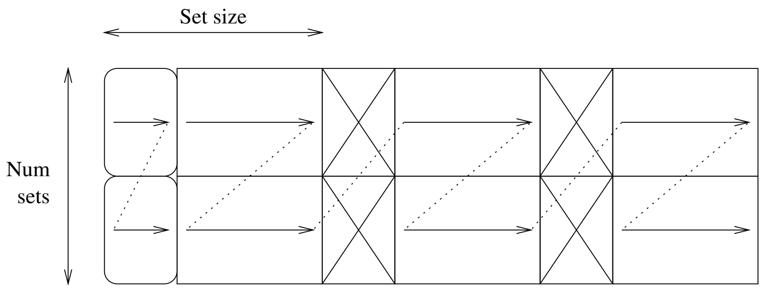

Figure 7: Sequence mapping for a set associative instruction cache.

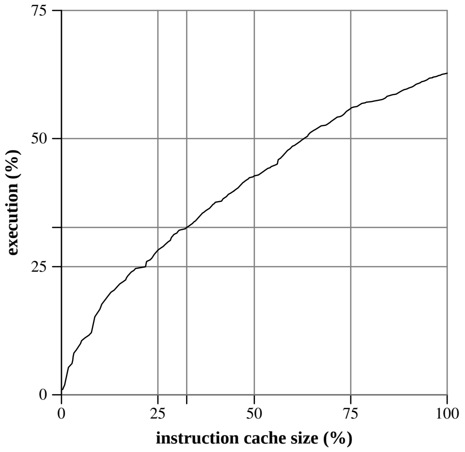

Figure 8: Illustration of the CFA sizing heuristic for an 8KB instruction cache. We fix the CFA size where execution percent and cache size usage meet.

instruction cache space. On the other hand, 25% of the instruction cache gathers 28% of the execution, leaving some room to shield more routines from interference.

4 Simulation results

We simulate an aggressive fetch unit as the one shown in Figure 1 with a variety of instruction cache and trace cache sizes for both the original code layout generated by the compiler and the optimized code layout obtained using the software trace cache. Table 1 provides a detailed description of the simulation setups examined.

Instruction cache	8KB to 64KB, 2-way set associative
Trace cache	none, 4KB, 16KB and 64KB, 2-way set associative, not path associative
Branch predictor	Gshare adapted to multiple predictions per cycle, 12 history bits, 4096 PHT entries
BTB	512 entries, 4-way set associative
RAS	64 entries

Table 1: Simulation setups examined for both the original and the optimized code layouts.

We simulate a different set of TPC-D queries[1] on a Btree indexed 100MB database using a complete fetch unit simulator derived from the SimpleScalar 3.0 tool set [2]. We simulate the reordered code by applying a

[1] See Appendix A

simple address translation, accounting for those unconditional branches that have to be added or removed. The simulator executes the original code but we collect statistics on the translated addresses.

4.1 Instruction cache miss rate

Figure 9 shows the number of first level instruction cache misses for several cache sizes (8 to 64KB), code organizations (original and STC reordered) and fetch unit models (base and with a 64KB trace cache). Note that the Y axis is in log_{10} scale due to the large differences between the examined cache configurations. The trace cache size is considered because on a trace cache hit, any instruction cache miss is ignored. The more instructions stored in the trace cache, the fewer misses will occur.

We present our results as the absolute number of instruction cache misses because the number of instruction cache accesses is not constant for all setups, only the number of instructions executed is. We only access each cache line once per fetch access, not once per instruction, and we ignore the instruction cache misses when we hit on the trace cache. The absolute number of misses allows a better comparison of the instruction cache performance.

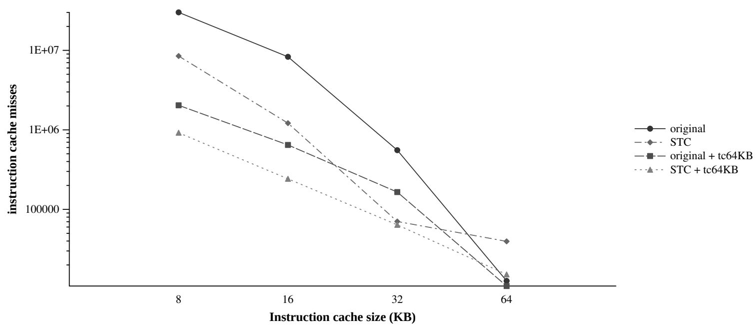

Figure 9: First level instruction cache misses for several instruction cache sizes, code organizations and fetch unit models (base and with a 64KB trace cache). On a trace cache hit, the instruction cache access is ignored.

When we compare the cache performance of the original code with the STC optimized one, it is clear that a well managed instruction cache can obtain results that we would associate with a much larger cache. For example, the STC optimized code has the same number of misses on an 8KB instruction cache as the original code on a 16KB one. An exceptional result shows that a well managed 32KB instruction cache holds the working set of PostgreSQL better than a 32KB instruction cache and a 64KB trace cache together using the original code layout.

As could be expected from the results in Figure 2, the 64KB instruction cache holds the working set of PostgreSQL with relative ease, no matter the code layout, reducing the number of misses to almost zero if we account for the cold misses.

Finally, it is worth noting the large influence of the 64KB trace cache in the smaller instruction cache setups, reducing the number of instruction cache misses in an order of magnitude for both the original and the reordered code layouts.

4.2 Sequence length increase

In order to evaluate the increase in spatial locality obtained with the STC reordering, we repeat the measurements in Section 2.2, but using the reordered executable.

First, we evaluate how successful the STC is at reversing the branch directions. Figure 10 classifies branches by the percentage of times they follow the taken path in the reordered code.

These results show that most branches change from taken to not taken. Now 87% of all branches are always not taken, while only 4% are always taken. With this, only an average of 7% of all dynamic branches are taken.

But, as we mentioned earlier, unconditional jumps, subroutine calls and returns and indirect jumps always break the execution sequence. This will reduce the benefits expected from such a reduction of taken branches in terms of sequence length increase. Figure 11 shows the number of sequences of a given length that the reordered PostgreSQL executes. Average sequence length is shown with a vertical line.

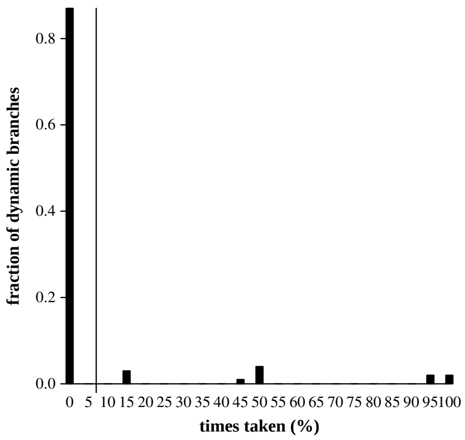

Figure 10: Percentage of branches classified by the number of times they follow the taken path in the reordered code. Average fraction of taken branches shown with a vertical line (7%)

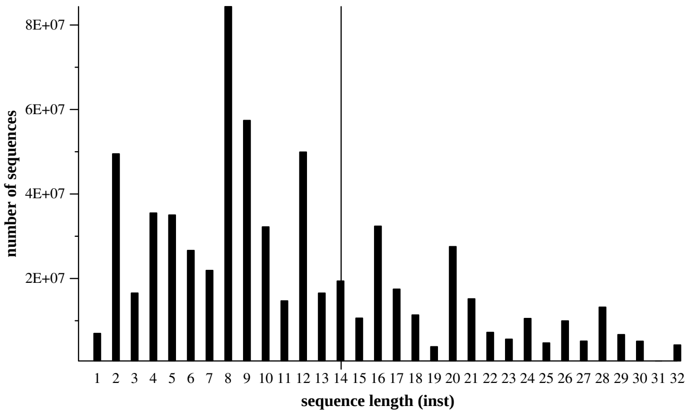

Figure 11: Number of times the reordered PostgreSQL executes a number of consecutive instructions between two taken branches/jumps. Average sequence length shown with a vertical bar.

As a consequence of the increase in the number of not taken branches, sequence length increases from 9.2 to 14 instructions. But the sequences are widely spread across different lengths, with many sequences being longer than 16 instructions, with a high peak at length 20. The main problem now is sequence fragmentation, because this high number of sequences longer than 16 instructions that will be broken by the hardware limit of 16 instructions.

4.3 Fetch performance

Figure 12 shows the number of instructions fetched per access to the fetch unit (FIPA) for both the original and the reordered code for a variety of trace cache sizes. This is a raw measure of the width of correct-path instructions provided each fetch cycle, not counting delays caused by cache misses and cycles wasted fetching from the wrong path. Because it is independent of the number of instruction cache misses, it is independent of the instruction cache size.

There is a large 30% increase in the number of instructions provided by the core fetch unit when we use the STC reordered code. As expected, this increase does not fully match the 50% increase we obtained in the average sequence length, due to sequence fragmentation. In any case, the fetch potential for the core fetch unit using the STC reordered code is the same as the fetch potential of the fetch unit complemented with a 64KB trace cache using the original code layout.

But the code reordering also combines well with the trace cache, continuing to obtain substantial performance increases with larger trace cache sizes. Thanks to the increased sequence length, the core fetch unit is able to provide more instructions on a trace cache miss. This backup improvement becomes less important as the trace

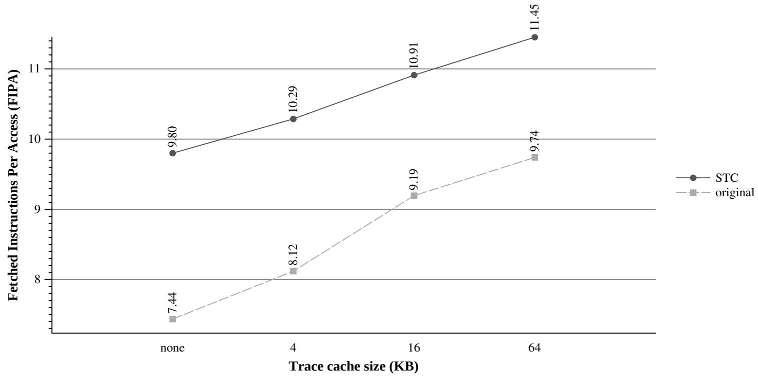

Figure 12: Number of fetched instructions per access to the fetch unit (FIPA) for the two code organizations and different trace cache sizes. These results are independent of the instruction cache size.

cache grows, because it will have fewer misses. Actually, the performance of the original code and the STC optimized one is expected to converge for infinite trace cache size.

But the benefits of using STC are not limited to an increased fetch potential. As we have seen, it also provides a significant reduction of the instruction cache misses, which will determine the number of cycles necessary to fetch those many instructions.

Figure 13 shows the number of instructions fetched per cycle (FIPC) for both the original and the reordered code for a variety of trace cache and instruction cache sizes. This measure accounts for the number of cycles it takes to fetch the instructions. We assume a uniform 6 cycle delay for each L1 instruction miss, and branch mispredictions are assumed to waste 12 cycles fetching from the wrong path. On a trace cache hit, L1 instruction cache misses are ignored.

We do not present results for branch mispredictions because both binaries proved to be equally predictable, being the STC reordered one slightly better.

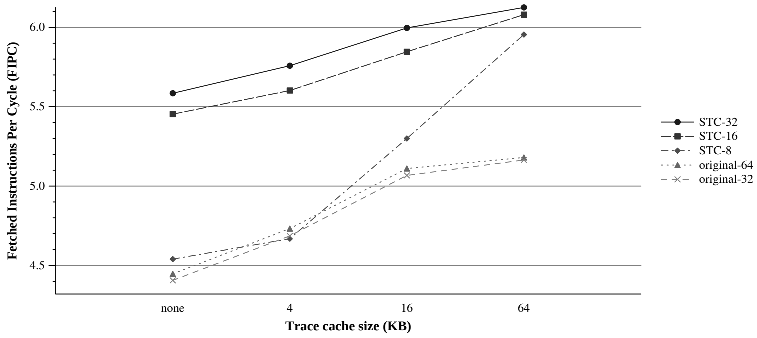

Figure 13: Number of fetched instructions per cycle (FIPC) for the two code organizations and different trace cache and instruction cache sizes.

When the delays caused by instruction cache misses and branch mispredictions are considered, the advantages of the STC become even more clear. A small 8KB cache with the reordered code obtains better fetch performance than much larger (32 and 64KB) caches with the original code layout.

For small to moderate trace cache sizes (4 and 16KB), the combination of a 8KB instruction cache and the STC obtains equivalent results to a 32 or 64KB instruction cache. The performance of the STC with larger instruction caches with the addition of a trace cache is much higher than that obtained with the original code layout.

For the largest trace cache setup (64KB), all optimized code layout setups (8, 16 and 32KB instruction cache sizes) obtain similar results. There are very few instruction cache misses left, and the number of branch mispredictions does not change with the instruction cache size, so all setups converge on the FIPA result obtained before (after branch misprediction penalties are applied). similar performance Meanwhile, the non-optimized code layout setups still have many instruction cache misses. Even with the 64KB instruction cache, which fits most of the working set, the small number of instructions provided by the core fetch unit on a trace cache miss still dominates the results.

We conclude that for large codes, with few loops and highly predictable execution sequences, the problem of fetching non-contiguous basic blocks is better approached by code reordering than with complex hardware mechanisms like the trace cache. But, our results also show that both techniques complement each other, and that higher performance can be obtained by combining both the software and the hardware approach.

5 Conclusions

In this paper we present a profile based code reordering technique which targets an optimization of the instruction fetch performance in the more aggressive wide superscalar processors.

By carefully mapping the basic blocks in a program we can store the more frequently executed traces in memory, using the instruction cache as a *Software Trace Cache* (STC), obtaining better performance of a sequential fetch unit, and complementing the Hardware Trace Cache (HTC) mechanism with a better failsafe mechanism.

Our results show that for large codes with few loops and deterministic execution sequences, like database applications, the STC can offer better results than the HTC alone [20]. However, optimum results come from the combination of both the software and the hardware approaches.

The storage of the most popular traces in the instruction cache leads to a new view of the fetch unit, where the trace cache is more tightly coupled with the contents of the instruction cache. Some traces are being redundantly stored in both caches, effectively wasting space, and displacing potentially useful traces from the trace cache. It is yet another example of the need for the software and the hardware to work together in order to obtain optimum performance with the minimum cost.

References

[1] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 3–14, June 1998.

[2] D. Burger, T.M. Austin, and S.Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report TR-1308, University of Winsconsin, July 1996.

[3] William Y. Chen, Pohua P. Chung, Thomas M. Conte, and Wen-Mei Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transactions on Computers*, 42(9):1045–1057, September 1993.

[4] T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. *Proceedings of the 22nd Annual Intl. Symposium on Computer Architecture*, pages 333–344, June 1995.

[5] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[6] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, December 1997.

[7] Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. Procedure placement using temporal ordering information. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, December 1997.

[8] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. *Proc. ACM SIGPLAN'97 Conf. on Programming Languaje Design and Implementation*, pages 171–182, June 1997.

[9] Derek L. Howard and Mikko. H. Lipasti. The effect of program optimization on trace cache performance. *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 256–261, October 1999.

[10] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Water, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara G. E. Haab, J. G. Hold, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal on Supercomputing*, (7):9–50, 1993.

[11] Wen-Mei Hwu and Pohua 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.

[12] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A thrid level of adaptivity for branch prediction. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 155–166, June 1998.

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

[14] Kimberly Keeton, David A. Patterson, Yong Quiang He, Roger C. Raphael, and Walter 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.

[15] M. Stonebreakerand G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, October 1991.

[16] Jack L. Lo, Luiz André Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay 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. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. *Proceedings of the 6th Intl. Conference on Architectural Support for Programming Languajes and Operating Systems*, pages 145–156, October 1994.

[18] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN'99 Conf. on Programming Languaje Design and Implementation*, pages 16–27, June 1990.

[19] Alex Ramírez, Josep Ll. Larriba-Pey, Carlos Navarro, Xavi Serrano, Josep Torrellas, and Mateo Valero. Optimization of instruction fetch for decision support workloads. *Proc. of the Intl. Conference on Parallel Processing*, pages 238–245, September 1999.

[20] Alex Ramírez, Josep Ll. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. *Proceedings of the 13th Intl. Conference on Supercomputing*, page to appear, June 1999.

[21] Parthasarathy Rangananthan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz André 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 Languajes and Operating Systems*, October 1998.

[22] E. Rottenberg, S. Benett, and J. E. Smith. Trace cache: a low latency aprroach to high bandwith instruction fetching. *Proceedings of the 29th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 24–34, December 1996.

[23] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *Proceedings of the 7th Intl. Conference on Architectural Support for Programming Languajes and Operating Systems*, October 1996.

[24] Josep Torrellas, Chun Xia, and Russell Daigle. Optimizing instruction cache performance for operating system intensive workloads. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 360–369, January 1995.

[25] Pedro Trancoso, Josep Ll. Larriba-Pey, Zheng Zhang, and Josep Torrellas. The memory performance of dss commercial workloads in shared-memory multiprocessors. *Proceedings of the 3rd Intl. Conference on High Performance Computer Architecture*, February 1997.

[26] T. Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *Proceedings of the 7th Intl. Conference on Supercomputing*, pages 67–76, July 1993.

A TPC-D query selection

To gain insight on the internal workings of PostgreSQL we identified the entry points to the different modules of the database. We instrument the database and insert code before and after each of those functions to count the number of instructions executed and the i-cache misses of each database module. The numbers obtained are reflected in the bottom row of Table 2 and show results for a direct mapped 32KB i-cache.

For a sample execution of all read-only queries on the Btree indexed database, a total of 169.5 billion instructions were executed, and 4.8 million cache misses were registered (a 2.8% base miss rate). As could be expected, most of the instructions belong to the query execution kernel. Less than 1% of the instructions belong to the user interface and the query optimizer levels, while 63% of the instructions belong to the Executor module. Nevertheless, the Access Methods and Buffer Manager modules account for 35% of the instructions, reaching 70% for some queries[2].

Looking at the i-cache misses, we observe that while the Executor is responsible of 63% of the executed instructions, only 53% of the misses correspond to that module. Meanwhile, the Access Methods gather 26% of the misses, for only 15% of the executed instructions. That is due to the fact that the Executor concentrates the few loops present in the database code, while the Access Methods are sequential functions, with few or no loops, and consist of many different functions referenced from several places which tend to conflict in the cache, replacing each other.

We were interested in learning which Executor operations were responsible for these instructions in the lower levels of the database. By modifying our instrumentation, we also counted the number of instructions and i-cache misses of each operation and the lower level functions called by those operations. We obtained the two dimensional matrix shown in Table 2. Dashes mean that no instructions were executed for that segment, while zeros represent an insignificant fraction of the total number of instructions.

	Other	Parser	Optimizer	Executor	Access	Buffer	Storage	Total
Other	0.2/0.0	0.0/0.0	0.7/0.0	0.1/0.1	0.1/0.1	0.0/0.0	0.0/0.0	1.2/0.3
Hash Join	–	–	–	0.0/0.0	–	–	–	0.0/0.0
Hash	–	–	–	0.0/0.0	–	–	–	0.0/0.0
Aggregate	–	–	–	0.2/0.2	0.0/0.0	0.0/0.0	0.0/0.0	0.2/0.2
Group	–	–	–	0.5/0.5	–	–	–	0.5/0.5
Sort	–	–	–	1.0/0.3	–	–	–	1.0/0.3
Merge Join	–	–	–	0.5/1.6	0.0/0.0	0.0/0.0	0.0/0.0	0.5/1.7
Nest Loop	–	–	–	1.4/1.4	0.2/0.4	0.0/0.0	0.0/0.0	1.6/1.8
Index scan	–	–	–	9.4/20.3	11.2/18.4	13.3/13.0	0.9/3.0	34.8/54.6
Seq. scan	–	–	–	2.8/3.1	3.7/7.3	6.2/3.8	0.0/0.0	12.7/14.3
Result	–	–	–	0.6/0.9	0.0/0.0	0.1/0.0	0.0/0.0	0.8/0.9
Qualify	–	–	–	46.8/25.3	0.0/0.0	0.0/0.0	0.0/0.0	46.8/25.3
Total	0.2/0.0	0.0/0.0	0.7/0.0	63.4/53.8	15.2/26.2	19.7/16.9	0.9/3.0	100.0/100.0

Table 2: Percentage of the total number of dynamic instructions/i-cache misses for each database level and each executor operation for a sample run of read-only queries on the Btree indexed database. Misses are for a direct mapped 32KB cache.

The most important operations are the Qualify operation and the Index and Sequential scan. The Qualify operation is responsible for checking whether a tuple satisfies a set of conditions or not, isolating all other Executor operations from the complexity of data types and operators. The Scan operations are responsible for most of the data access in the query execution. Indeed, almost all the references to the Access Methods and the Buffer

[2]Per query numbers not shown.

Manager belong to the Scan operations. The Sequential scan makes heavier use of the Qualify operation as it must check all tuples in the scanned table, while the Index scan needs to check fewer tuples, because only those tuples that satisfy the index condition will be accessed.

The Index scan is responsible of 54% of the misses for only 34% of the instruction references. Most of the misses due to the Index scan are found in the Executor and the Access Methods modules. The Index scan operation accounts for so many misses due to its irregular behavior, using many different Access Methods routines to access both indices and database tables. Meanwhile, the Sequential scan has fewer misses, because it only accesses database tables, using less Access Methods routines.

On the other hand, the Qualify operation is responsible for 25% of the misses, while it gathers as much as 46% of the executed instructions. Its heavy use, and the repeated evaluation of the same conditions across a given query make it easy for the cache to hold its working set.

We conclude that the Executor module, and the Qualify and Scan operations in particular, concentrate most of the executed instructions. Also, the Access Methods and Buffer Manager modules must be taken into account as they concentrate a large percentage of the total i-cache misses.

Based on this data, we select queries 3, 4, 5, 6, 9 and 15 as a representative set to be used for profile feedback optimizations, and queries 2, 3, 4, 6, 11, 12, 13, 14, 15 and 17 as our reference input set. Queries 1, 7, 8, 10 and 16 were not included because they take much longer to execute, and do not exercise the fetch mechanism in any way that is not present in the reference queries.