Trace cache redundancy: Red & Blue traces *

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

Abstract

The objective of this paper is to improve the use of the hardware resources of the trace cache mechanism, reducing the implementation cost with no performance degradation. We achieve that by eliminating the replication of traces between the instruction cache and the trace cache.

As we show, the trace cache mechanism is generating a high degree of redundancy between the traces stored in the trace cache and those built by the compiler, already present in the instruction cache. Furthermore, code reordering techniques like the software trace cache arrange the basic blocks in a program so that the fall-through path is the most common, effectively increasing this trace redundancy.

We propose selective trace storage to avoid trace redundancy between the trace cache and the instruction cache. A simple modification of the fill unit allows the trace cache to store only those traces containing taken branches, which can not be obtained in a single cycle from the instruction cache.

Our results show that selective trace storage and the software trace cache used on a 32 entry trace cache (2KB) perform as well as a 2048 entry trace cache (128KB) without the enhancements. This shows that the cooperation between hardware and software is crucial to improve the performance and reduce the requirements of hardware mechanisms in the fetch engine.

1. Introduction

The capability of future wide issue superscalars to execute large numbers of instructions per cycle is going to put an extra pressure on their fetch mechanism. Multiple basic block per cycle will be fetched, which adds two new factors to fetch performance: instruction sequentiality and multiple branch prediction.

These two problems can be solved using both software and hardware approaches. On the hardware side we find techniques like the branch address cache [18], the collapsing buffer [2] or the trace cache [4, 9, 16]. On the software side, we find instruction scheduling techniques [3, 7], and code reordering approaches like the *software trace cache* (STC) [11, 13, 14].

It has been shown that both hardware and software approaches combine well to obtain improved results [13, 14] with trace caches of half the original size. This combination allows a cost reduction in the fetch unit implementation without any performance degradation.

Based on this fruitful collaboration of compile-time and run-time techniques, we focus on a further reduction of this hardware implementation cost. We analyze the instruction stream from the trace cache point of view, and find significant redundancy in it. Some traces are already generated at compile-time, and these traces are being stored in both the instruction cache and the trace cache. The number of compile-time generated traces increases when we improve the code layout. Our results show that after reordering the code with the STC, over 60% of the issued traces do not contain any sequence break, and can be issued in a single cycle from an aggressive sequential fetch unit without need of a trace cache.

We propose a modification of the fill unit to implement *selective trace storage* (STS), that is, avoid storage of traces consisting of consecutive instructions. With STS and STC on very small trace caches, we obtain better performance than using large caches without those techniques.

We obtain better performance with a 2KB trace cache using a combination of STS and STC than a 128KB trace cache with none of our enhancements.

*This work was supported by the Ministry of Education and Science of Spain under contract TIC-0511/98 and by CEPBA. Alex Ramirez is also supported by Generalitat de Catalunya grant 1998FI-003060-26. The authors want to thank the anonymous reviewers for their insightful comments.

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: instruction cache misses, branch mispredictions, 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.

1.2. Related work

The branch address cache [18] and the collapsing buffer [2] mechanisms target the problem of fetching discontinuous basic blocks in a single cycle. Both include a complex hardware mechanism to package instructions from different instruction cache lines in the fetch stage, before passing them to the decode stage. Such a complicated process may affect the cycle time of the processor, or add an extra pipeline stage.

The trace cache [4, 9, 16] comes after the previous two mechanisms and moves the instruction packing process out of the execution pipeline, storing the built traces in a special purpose *trace cache*. This effectively avoids the cycle delay problem, but requires extensive additional hardware in the form of a special purpose cache memory.

The paper *Putting the fill unit to work* [5] considers the fill unit as more than just an instruction packager. Authors propose to use the off-line time to do extra optimization work on the instructions in the trace, like explicitly marking register move operations, collapsing of immediate values and instruction scheduling to minimize latency through the bypass network.

On the software side, trace scheduling techniques point that basic blocks execute in an almost deterministic order, and define algorithms to group them in basic block traces [3] or closed execution zones (superblocks and hyperblocks) [7]. These techniques define a logical ordering of basic blocks and reorder instructions in those logical groups crossing the basic block boundary to optimize instruction scheduling for VLIW processors.

There are also other code reordering algorithms which target an increase in the instruction cache hit rate [6, 10, 17]. In order to exploit more spatial locality, these techniques also increase the code sequentiality, but they target a single basic block fetch per cycle. The full potential of those techniques has not been exploited.

The *software trace cache* (STC) [11, 12, 13, 14] is the first code reordering targeting the more aggressive processors, and pointing that instruction fetch is better approached using both software and hardware techniques. Targeting a better usage of the core fetch unit, the STC builds instruction traces at compile-time, increasing code sequentiality and reducing the instruction cache miss rate. Using STC, we increase the performance of the core fetch unit, reducing the need for the trace cache, and obtaining similar or better performance with half to a fourth the trace cache instruction storage.

The block-based trace cache [1] is an alternative trace cache organization which also targets a cost reduction. It avoids redundancy between the basic blocks composing the different traces by storing them in a special purpose replicated block cache, and then builds traces by storing pointers to these basic blocks in a trace table. We examine this trace cache organization more closely in [15], applying STS to it, and proposing a modification to reduce unnecessary block cache replication.

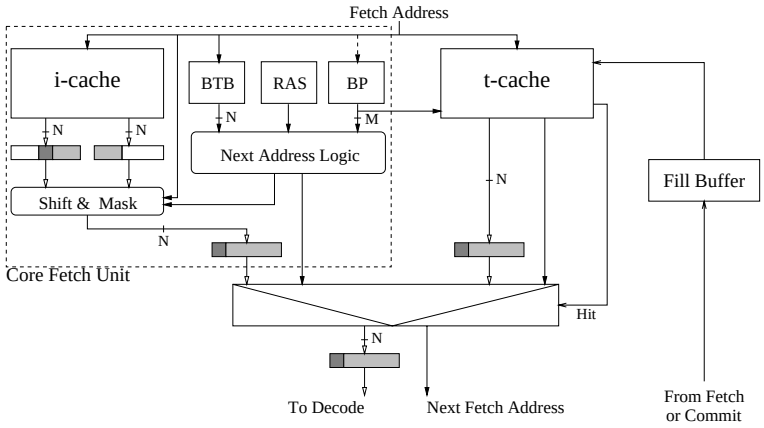

Figure 1. Core fetch unit able to fetch instructions from multiple consecutive basic blocks in a single cycle. Extension of the core fetch unit with a trace cache to allow fetching of non-consecutive basic blocks.

1.3. Paper structure

This paper is structured as follows. Section 2 analyzes the instruction stream from the trace cache point of view, and quantifies the trace redundancy between the instruction cache and the trace cache. In Section 3 we propose *selective trace storage* in order to avoid this trace redundancy. Section 4 presents performance improvements obtained by STS for a number of trace cache and instruction cache sizes, as well as for realistic and perfect branch prediction, using both the original code and an optimized code layout. Finally, in Section 5 we present our conclusions and guidelines for future work.

2. Trace cache redundancy

The trace cache mechanism is generating a certain degree of redundancy between the trace cache and the instruction cache. It is not just that both caches are storing the same instructions: some instruction sequences are replicated in both caches.

An instruction sequence containing no taken branches can be fetched from the instruction cache in a single cycle without need of the trace cache. But, as shown in Figure 2, such traces are also stored in the trace cache.

We will distinguish between two types of traces: blue traces, which contain only consecutive instructions, and red traces, which contain some form of sequence break (taken branch or unconditional jump).

Red traces are built by the fill unit at run-time, and stored in the trace cache, while blue traces are built by the compiler, and stored in the instruction cache. However, the fill unit also captures these blue traces, and stores them in the trace cache, effectively replicating them.

Code reorderings like the *software trace cache* (STC) [14] arrange the basic blocks in a program so that the most likely execution path does not contain any taken branch. It does so by moving basic blocks so that the non-taken branch target is the most likely, and including unused basic blocks after the main execution path has been completed. This reduction in the number of sequence breaks found during program execution increases the proportion of blue traces.

Table 1 shows a breakdown of the dynamic trace stream grouped by the number of sequence breaks they contain: from zero to three or more breaks. The first vertical half of the table shows the numbers for the original code layout, and the second half corresponds to the STC reordered code.

There is a high proportion of blue traces (traces containing 0 breaks), even in the original code layout: over 70% for FP codes and around 40% for integer applications. This shows that there is a representative degree of redundancy between the traces stored in the trace cache and what the core fetch unit can provide in a single fetch unit access.

Reordering the basic blocks increases the blue trace proportion to 51–88% for integer applications, effectively reducing the average number of breaks found in a trace and increasing the trace redundancy.

Reordering the code not only generates more blue (consecutive) traces, it reduces the number of breaks significantly (as shown in Table 1). This reduces the number of cycles required to build a red trace from the core fetch unit, reducing the need of a support mechanism like the trace cache, or providing a better fail-safe mechanism in case of a trace cache miss.

It is important to note that any other basic block reordering technique such as [6, 10, 17] would produce a similar effect to that observed in Table 1. A comparison be-

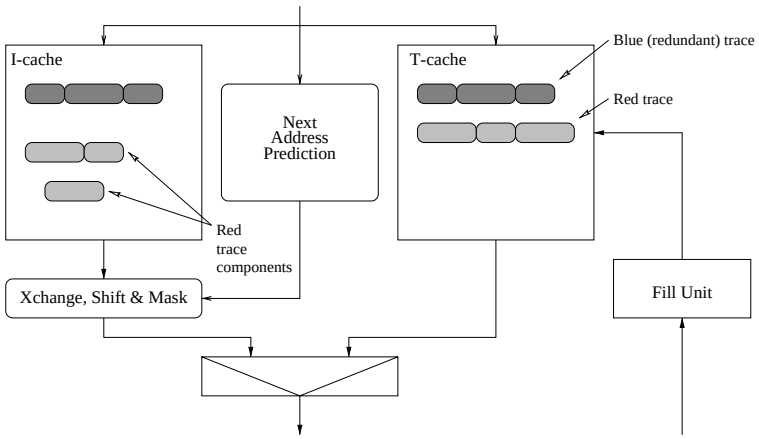

Figure 2. Redundancy of traces between the instruction cache and the trace cache. Traces consisting entirely of consecutive instructions (blue traces) can be fetched from the instruction cache in a single cycle without a trace cache.

tween [10, 17] and the STC can be found in [11, 13].

3. Selective Trace Storage

A simple extension of the fill unit logic would allow it to distinguish between red and blue traces. Only red (discontinuous) traces would be stored in the trace cache. With this simple modification we can avoid replacing the red traces from the trace cache with a blue one that we could obtain from the core fetch unit, allowing a smaller trace cache to accommodate the same number of red traces.

This modification will not have an important impact on the cycle time of the processor, as the logic complexity in the fill buffer will increase moderately. If that was the case, the fill buffer could be pipelined, increasing the time it takes for a new trace to reach the trace cache, but previous results in [8] show that there is little performance impact due to the fill unit latency.

The same number of trace entries should store more red traces than before, as the blue traces are not using up any trace cache space. The global trace cache miss rate will go up, as blue traces will always cause a trace cache miss, but the chances of finding the desired red trace go up, and blue traces will still be available from the instruction cache. On the other hand, the same number of red traces can be captured with less trace cache entries, reducing the trace cache implementation cost.

4. Evaluation

Selective trace storage (STS) targets an increase in the number of instructions provided by the fetch unit each time it is accessed. For this reason, we evaluated the performance of our fetch unit as a stand-alone block, avoiding interference from the execution core of the processor. An increase in the fetch bandwidth will be beneficial to any architecture.

4.1. Simulation setup

Table 2 describes all the simulation setups explored. Each PHT entry in the Gshare branch predictor has three separate two-bit counters, one for each of the three predictions provided. Each branch is tagged with the counter address to update. While it is possible to find better branch predictors than the one we used for this paper, this one is relatively small, and easily implementable, matching the cost reduction effort of the paper.

An increase in the fetch bandwidth will be beneficial to any architecture, but the final IPC performance depends on many other architecture parameters.

For this reason, results are presented in terms of Fetched Instructions Per Access (FIPA) which measures the number of correct-path instruction provided to the execution core each time the fetch unit is accessed, and in terms of Fetched Instructions Per Cycle (FIPC) which also accounts for the number of cycles per fetch unit access. We assumed a 6 cycle delay for each instruction cache miss, and a 12 cycle delay for each branch or target misprediction.

Remember that FIPA results do not account for the stall cycles due to instruction cache misses. This means that FIPA results do not depend on the instruction cache size. Meanwhile, the trace cache miss rate determines the length of the instruction traces provided, which does affect the FIPA.

The SPEC benchmarks were simulated to completion using the *ref* input set, and a representative subset of the TPC-D queries [11, 13] on a 100MB database was used for *Post-*

	Original				STC Reordered			
Bench.	0	1	2	3+	0	1	2	3+
tomcatv	75	23	1	0	77	23	0	0
swim	71	29	0	0	71	29	0	0
su2cor	67	29	4	0	71	25	2	2
hydro2d	58	28	14	0	69	27	3	1
mgrid	78	21	0	0	78	21	0	0
applu	69	27	4	0	70	24	6	0
turb3d	78	20	2	0	82	17	1	0
apsi	67	31	2	0	72	27	2	0
fpppp	95	4	1	0	93	6	1	0
wave5	69	28	3	0	71	27	2	0
FP avg	73	24	3	0	75	23	2	0
m88ksim	37	36	25	2	67	30	2	0
gcc	33	42	22	2	51	35	13	1
compress	44	33	20	3	46	39	10	5
li	35	35	26	3	49	39	11	1
ijpeg	61	21	18	0	65	21	10	4
perl	34	45	20	1	60	30	8	1
vortex	37	49	13	0	88	9	3	0
INT avg	40	37	21	1	61	29	8	1
postgres	38	40	21	1	66	23	10	1

Table 1. Distribution of traces classified by the number of sequence breaks they contain. Numbers shown for both the original code layout and the STC reordered code. Traces with 0 breaks are considered blue traces.

gres. We used the *train* input set to obtain the profile information for the SPEC benchmarks, and a different set of TPC-D queries for *postgres.*

We only present simulation results for *gcc* and *li* from the SPEC benchmarks, and *postgres*, a relational database management system, running the TPC-D benchmark.

Floating point codes already obtain a very high fetch bandwidth without trace cache, and *ijpeg* behaves like a FP code, having large basic blocks and lots of loops. It does not require a trace cache for high fetch performance. On the other hand, *perl* behaves very much like *li* but is much smaller, requiring even less trace cache storage. Also, *vortex* and *m88ksim* behave in a similar way to *postgres* but are much smaller. Finally, the instrumented version of *go* crashed for unknown reasons.

To reduce the required simulation time, we obtained detailed simulation results for 5% of the executed instructions (detailed simulation of 25 million instructions every 500 million). We have verified (for shorter input sets) that this sampling simulation obtains equivalent results to the full simulation.

Instruction cache	32KB, 64 Byte lines, direct mapped **or** 64KB, 64Byte lines, 2-way set associative
Trace cache	32 to 2048 entries, 2-way set associative
Branch predictor	Gshare adapted to multiple predictions per cycle, 12 bits of history, 4096 PHT entries, saturating 2-bit counters **or** perfect branch prediction
Branch target buffer	512 entries, 2-way set associative **or** perfect target prediction
Return address stack	perfect

Table 2. Simulation setups explored

4.2. Realistic branch prediction

Figure 3 shows FIPA performance results for trace cache sizes from 32 to 2048 entries (2KB to 128KB of instruction storage) with and without STS using the realistic branch predictor described above. Remember that FIPA performance does not depend on the instruction cache size. Setups tagged TC use the original code layout and setups tagged STC use the optimized layout. Setups tagged with (+) use STS.

Comparing vertical points in two plots we appreciate the performance improvements obtained using STC (TC vs STC) and STS (TC vs TC+ and STC vs STC+). As we already knew from [14] there is a large performance improvement when we use STC. STS adds a little extra performance, but not as large as STC.

Instead of comparing vertical performance points, we compare horizontal points, which show the trace cache size reduction that STS can obtain. In general terms, a STS-enhanced trace cache obtains the same performance as a non-enhanced one of double size. For example, a 512-entry trace cache with STS (TC+.512) obtains the same performance as a non-enhanced 1024-entry one (TC.1024) for all benchmarks.

As we have shown in Table 1, STS is reducing the number of traces stored in the trace cache from 33% to 38% for the selected benchmarks (blue traces are not stored in the trace cache anymore). This is increasing the effective trace cache size in an equivalent percentage.

The redundancy between the instruction cache and the trace cache increases to 49%–66% for the STC optimized layout, leading to increased benefits of the use of STS. Using both STC and STS we can improve on the FIPA performance of a trace cache of double to four times the size (STC+.64 *vs* STC.128 and STC.256).

It is important to note that all curves converge as we increase the trace cache size, obtaining equivalent perfor-

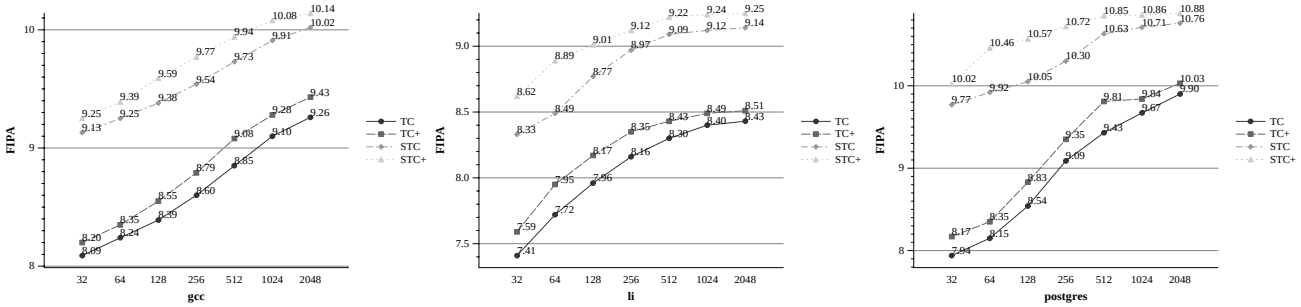

Figure 3. FIPA performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) using realistic branch prediction. The STS enhanced models are tagged with (+). Setups labeled STC use the optimized code layout.

mance for an infinite number of lines. As programs fit more and more comfortably in the cache, there is less benefit in increasing the cache size, or using STS, which produces a similar effect. We propose STS as a cost reduction technique, not a performance improvement, as it allows to reach a given performance level using less storage space.

Fetched Instructions Per Cycle

Figure 4 shows FIPC performance results for trace cache sizes from 32 to 2048 entries (2KB to 128KB of instruction storage) with and without STS using the realistic branch predictor. Separate graphs are provided for FIPC performance on instruction caches of 32 and 64KB. Setups tagged TC use the original code layout and setups tagged STC use the optimized layout. Setups tagged with (+) use STS.

FIPC results account for the stall cycles caused by instruction cache misses and the wasted cycles due to branch mispredictions, dividing the FIPA performance obtained. The influence of these extra cycles hides most of the benefits of using STS. Performance actually decreases for *gcc* and *postgres* (the two largest codes examined) when a very large trace cache is used (1024 or 2048 lines), why?

STS does not store blue traces in the trace cache assuming that they will be available from the instruction cache, but this is not always the case. As blue traces are always obtained from the instruction cache, the total number of accesses increases, leading to more instruction cache misses and more stall cycles. The FIPA increase allows the fetch unit to provide the same number of instructions in less fetch unit accesses, but if we increase the number of cycles for each access, it can result in a decreased FIPC performance.

We can avoid this effect using any technique which reduces the instruction cache miss rate, like code reorderings and larger/more associative caches. This can be observed comparing the curves for 32 and 64KB caches and the curves for the original and the optimized code layouts (TC vs STC).

Using a combination of STC and STS, we obtain better FIPA performance with a 32-entry trace cache (2KB) than a 2048-entry one (128KB) without any of our enhancements. Depending on the instruction cache miss penalty, and the branch misprediction delay, this translates into equivalent FIPC performance.

While the use of STS and a set-associative 64KB cache reduced the instruction cache misses to almost zero, the FIPC obtained is still much lower than the FIPA. This is due to branch and target address mispredictions, which cause the fetch unit to waste cycles fetching instructions from an incorrect execution path. For this reason, we also examined the effect of STS using perfect branch prediction.

4.3. Perfect branch prediction

Now, we examine the limits of STS under perfect branch prediction. Figure 5 shows FIPA performance results for trace cache sizes from 32 to 2048 entries (2KB to 128KB of instruction storage) with and without STS, using perfect branch and target prediction. FIPA performance does not change from a 32KB to a 64KB instruction cache. Again, setups tagged TC use the original code layout and setups tagged STC use the optimized layout. Remember that FIPA results do not depend on the instruction cache size.

The benefits of STS are more evident in the presence of a perfect branch predictor. We obtain the same performance with a trace cache of half to a fourth the size, both with the original code layout and the STC optimized layout. The benefits of STS are more clear for the optimized code layouts and the smaller trace cache sizes.

Using a perfect branch predictor, the convergence of all setups for increasing trace cache sizes is more clear. The baseline configuration increases performance faster than the STS+STC setup does, and they eventually reach the same

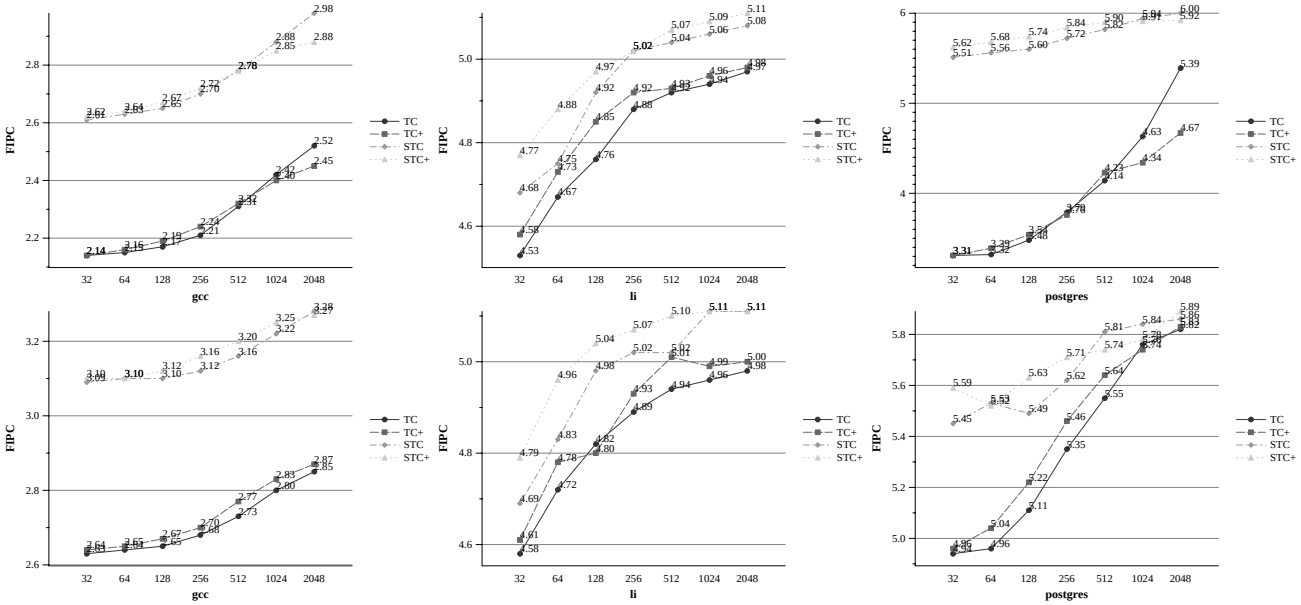

Figure 4. FIPC performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) and instruction cache sizes of 32 and 64KB using realistic branch prediction. The STS enhanced models are tagged with (+). Setups labeled STC use the optimized code layout.

performance for infinite trace cache size. The benefits of using STS and STC decrease as the trace cache size increases. Remember that we propose the use of STS and STC as a way of reaching peak performance with the minimum cost, not as a way to increase peak performance.

Figure 5 also shows an unexpected result: STS alone provides better FIPA performance than the use of STC for the larger caches (TC.1024+ *vs* STC.1024), which did not happen with the realistic branch predictor. The following paragraphs explain this effect, based on the different length of red and blue traces.

Bench.	Trace Length: All traces	orig: Blue	orig: Red	STC: Blue	STC: Red
gcc	12.60	10.42	13.68	11.38	13.87
li	12.03	7.52	14.44	9.51	14.42
postgres	11.89	9.84	13.80	10.83	14.47

Table 3. Average dynamic trace length. Separate results for blue trace length and red trace length are provided for the original code layout and the STC optimized one.

Trace length

Perfect branch prediction increases the effective length of the provided traces, because no instructions in a trace belong to a wrong execution path. This is exposing a difference in the length of red and blue traces.

Table 3 shows the average trace length for the studied benchmarks. Blue and Red trace length are calculated separately for both the original and the STC reordered code layouts.

As expected, red traces are much longer than blue traces. STS does not store the blue (shorter) traces in the trace cache, which increases the number of instructions provided on a trace cache hit (if there is a hit, it is to a red/long trace). At the same time, storing only red traces increases the chances of a hit when the desired trace is a red one. This trace cache FIPA increase represents a larger fraction of the global FIPA for the largest trace cache setups, leading to the observed performance increases.

Meanwhile, the STC increases the FIPA of the core fetch unit only, leaving the trace cache crowded with both red and blue (shorter) traces. With perfect branch prediction, the core fetch unit FIPA increase is not as large as the trace cache FIPA increase produced by STS.

This effect is not visible with realistic branch prediction because branch mispredictions prevent the trace cache from providing whole correct traces. That is, not all instructions in the provided trace belong to the correct execution path, which reduces the effective trace length to nearly the blue trace length.

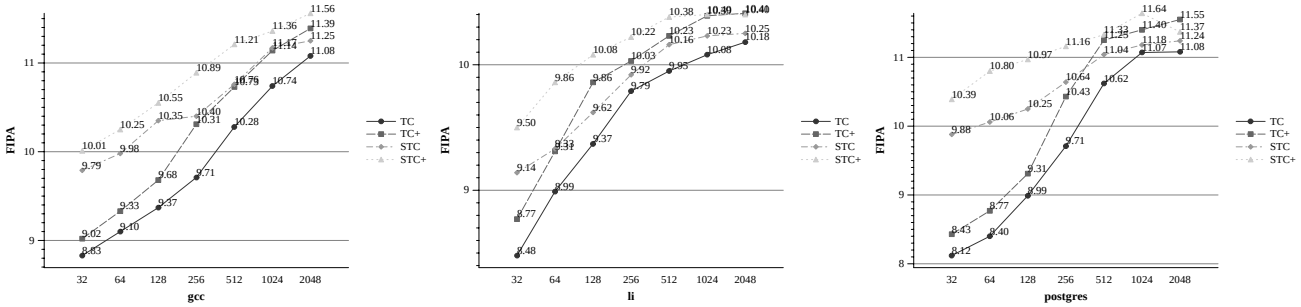

Figure 5. FIPA performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) using perfect branch prediction. The STS enhanced models are tagged with (+). Setups tagged STC use the optimized code layout.

Fetched Instructions Per Cycle

Figure 6 shows FIPC performance results for trace cache sizes from 32 to 2048 entries (2KB to 128KB of instruction storage) with and without STS, using perfect branch and target prediction. Setups tagged TC use the original code layout and setups tagged STC use the optimized layout. Separate graphs are provided for performance on instruction caches of 32 and 64KB.

Again, the impact of the increased instruction cache miss rate decreases FIPC performance on the largest trace cache configurations and the 32KB instruction cache (TC.2048+ *vs* TC.2048, and STC.2048+ *vs* STC.2048). This effect is not visible on the 64KB instruction cache due to the lower miss rate of the larger/more associative cache.

Having eliminated the interference of the branch predictor, we observe that the use of STS produces FIPC improvements, matching the FIPA performance obtained. Using a combination of STC and STS we can obtain equivalent performance to a non-enhanced trace cache eight to sixteen times larger (STC.32+ *vs* TC.512 on gcc and postgres, STC.128+ *vs* TC.2048 on li).

Once more, all setups offer similar performance for the largest trace cache sizes. STS and STC allow us to obtain near-optimum performance with fewer hardware cost, and provide less performance improvement as the trace cache grows.

5. Conclusions

We have shown that the compiler already generates a number of consecutive (blue) instruction traces, and that this number can be substantially increased using code reordering techniques like the *software trace cache* (STC). These blue traces are being stored in both the instruction cache and the trace cache, creating an unnecessary degree of redundancy.

We propose *selective trace storage*, a simple modification of the trace cache fill unit which avoids storage of these blue (redundant) traces. By not repeating at run-time the work that was done at compile-time, we obtain substantial hardware cost reductions.

Using STS we can obtain similar or better performance with half to a fourth the storage space, a cost reduction which adds to what was already obtained using code reordering techniques like STC. Using a combination of STC and STS we obtain better performance with a very small 32-entry trace cache than a 2048-entry one without any of the improvements using a realistic branch predictor.

Our results also show that the use of STS increases the pressure on the instruction cache, which makes using code reordering techniques more necessary, in order to keep the miss rate as low as possible.

As a final conclusion, we have shown that a tight cooperation between hardware and software techniques imply a substantial increase in performance and a considerable cost reduction of hardware devices.

References

[1] B. Black, B. Rychlik, and J. P. Shen. The block-based trace cache. *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, May 1999.

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

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

[4] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 1997.

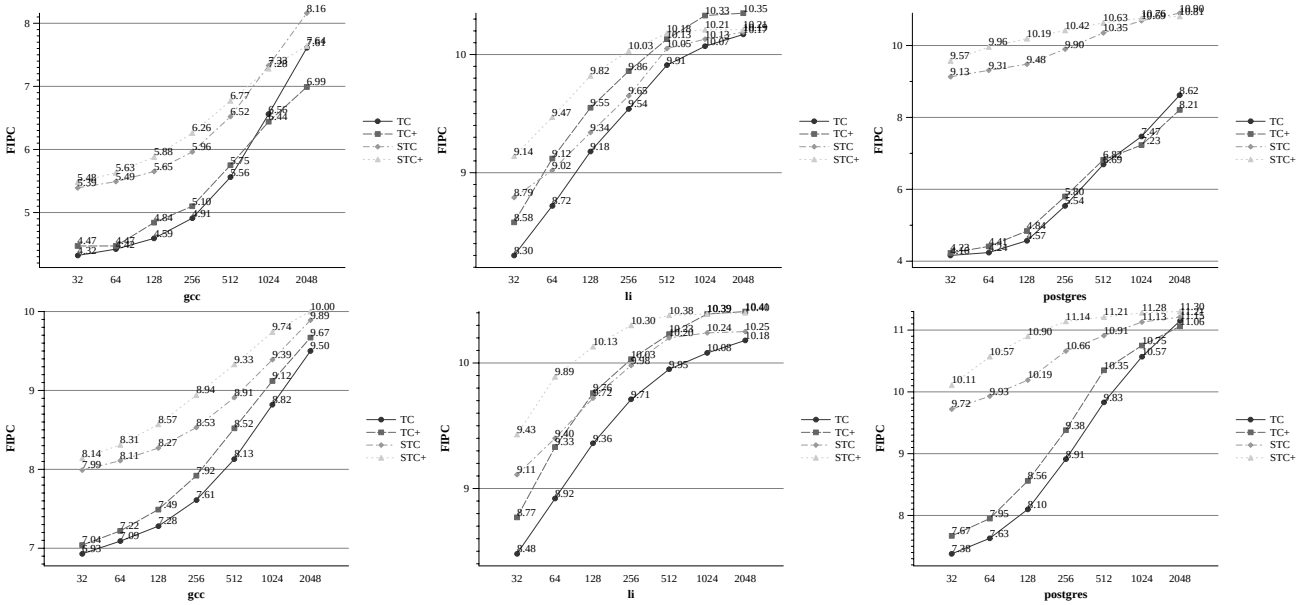

Figure 6. FIPC performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) and instruction cache sizes of 32 and 64KB using perfect branch prediction. The STS enhanced models are tagged with (+). Setups tagged STC use the optimized code layout.

[5] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimization for trace cache microprocessors. *Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 173–181, Nov. 1998.

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

[7] 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. K. 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.

[8] S. J. Patel, D. H. Friendly, and Y. N. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, University of Michigan, May 1997.

[9] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized arround trace segments independent of virtual address line. *U.S. Patent Number 5.381.533*, Jan. 1995.

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

[11] A. Ramírez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. Technical Report UPC-DAC-1998-56, Universitat Politecnica de Catalunya, Dec. 1998.

[12] A. Ramírez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero. Code reordering of decision support systems for optimized instruction fetch. *2nd Worksop on Computer Architecture Evaluation using Commercial Workloads*, Jan. 1999.

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

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

[15] A. Ramirez, J. L. Larriba-Pey, and M. Valero. Red & blue traces: Trace cache redundancy. Technical Report UPC-DAC-1999-26, Universitat Politecnica de Catalunya, June 1999.

[16] 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, Dec. 1996.

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

[18] 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.