5 SELECTIVE TRACE STORAGE

Once we have obtained an optimized application, and analyzed the reasons for the performance improvements obtained, we focus on a cost reduction of the underlying hardware, taking into account the advantages exposed by the compiler optimizations.

In the previous chapter we have described the software trace cache (STC) layout algorithm. The STC maps sequentially executed basic blocks to consecutive memory locations, making most branches biased towards their not taken direction. The trace cache is doing exactly the same task: mapping sequentially executed blocks to consecutive storage, only it is doing so at run-time.

As we show in this chapter, 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. The objective of this work is to improve the use of the hardware resources in the trace cache mechanism, reducing the implementation cost with no performance degradation.

We propose *selective trace storage* to avoid trace level 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 these enhancements.

5.1 Introduction

In Section 4.2.2 we have shown that both hardware (the trace cache) and software (the *software* trace cache) approaches combine well to obtain improved results with trace caches less than 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.

5.2 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 5.1, such traces are also stored in the trace cache.

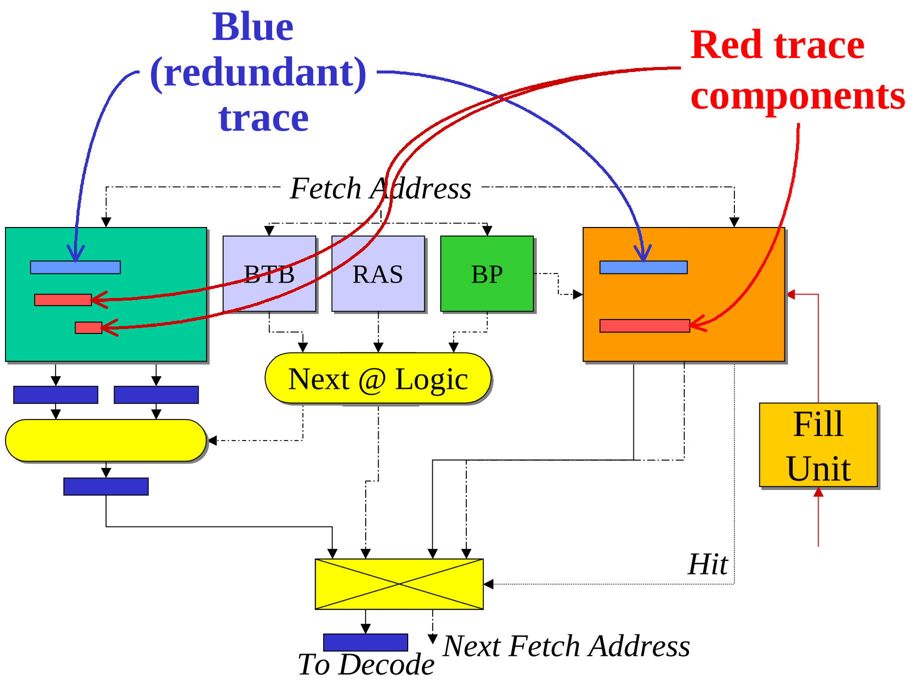


Figure 5.1. Redundancy of traces between the instruction cache and the trace cache. Traces consisting entirely of consecutive instructions (blue traces) can be fetched from the instruction cache in a single cycle without a trace cache.

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) 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.

Figure 5.2 shows a breakdown of the dynamic trace stream grouped by the number of sequence breaks they contain: from zero to three or more breaks.

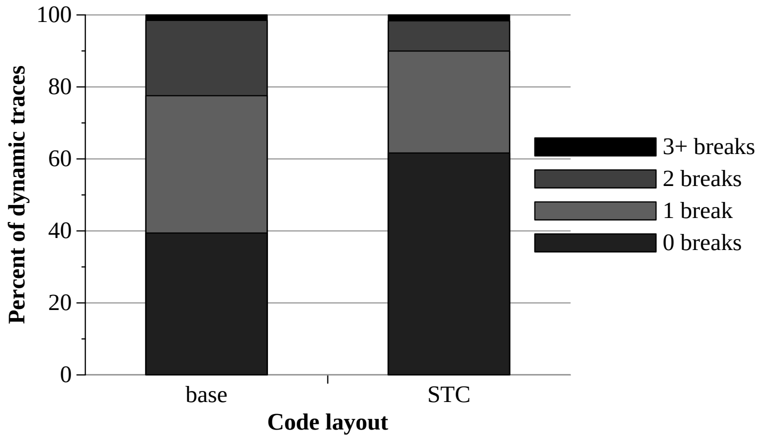


Figure 5.2. Distribution of traces classified by the number of sequence breaks they contain. Numbers shown for both the original code layout and the STC reordered code. Traces with 0 breaks are considered blue traces.

There is a high proportion of blue traces (traces containing 0 breaks), even in the original code layout: around 40% of all traces contain no taken branches in 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% (average 60% shown in the graph), effectively reducing the number of breaks found in a trace and increasing the trace redundancy.

Reordering the code not only generates more blue (consecutive) traces, it also reduces the number of breaks significantly. 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. For example, after optimization, 85% of all traces at most one taken branch, which means that they could be provided by the instruction cache in at most two cycles.

It is important to note that any other basic block reordering technique such as [32, 59, 87] would produce a similar effect to that observed in Figure 5.2.

5.3 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 [55] 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.

5.4 Evaluation

In this section we evaluate the benefits of selective trace storage (STS) in the front-end engine in realistic conditions, that is, in the presence of a realistic branch predictor.

Our results show that branch mispredictions are hiding most of the potential improvements, so we also evaluated STS under ideal conditions using a perfect branch predictor, which unveils an unexpected effect caused by red traces being longer than blue traces.

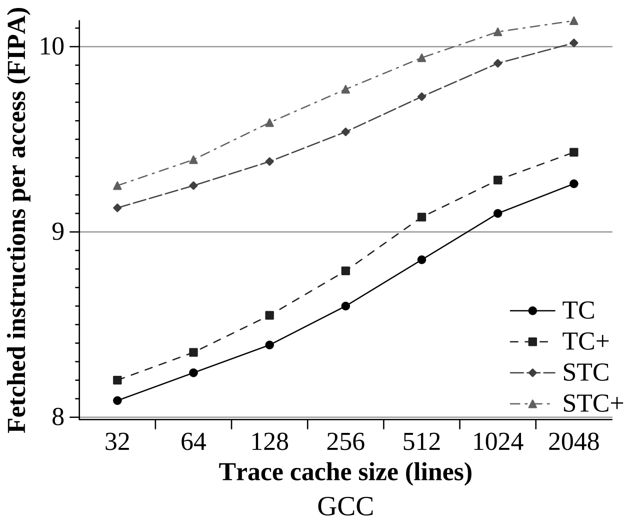
5.4.1 Realistic branch prediction

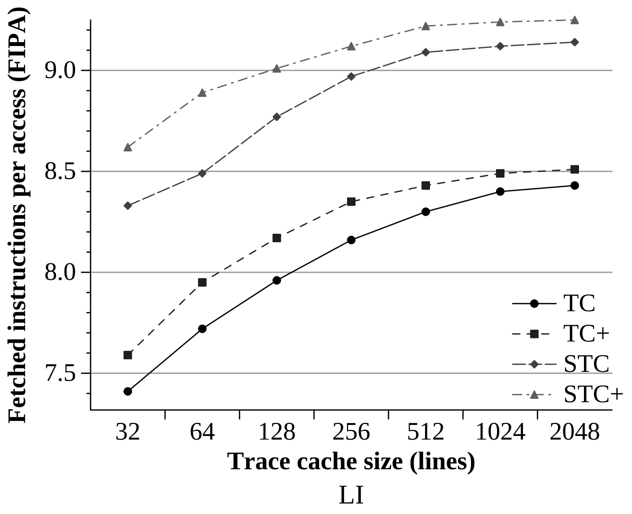
Figure 5.3 shows FIPA performance results for trace cache sizes from 32 to 2048 entries (2KB to 128KB of instruction storage) with and without STS using the realistic branch predictor described above. It is important to note that FIPA performance does not depend on the instruction cache size. Setups 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 Chapter 4 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 Figure 5.2, 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.





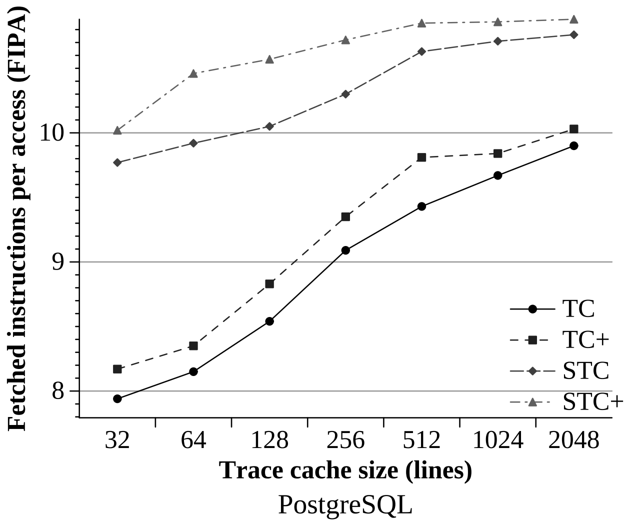


Figure 5.3. FIPA performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) using realistic branch prediction. The STS enhanced models are tagged with (+). Setups labeled STC use the optimized code layout.

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 performance 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 5.4 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.

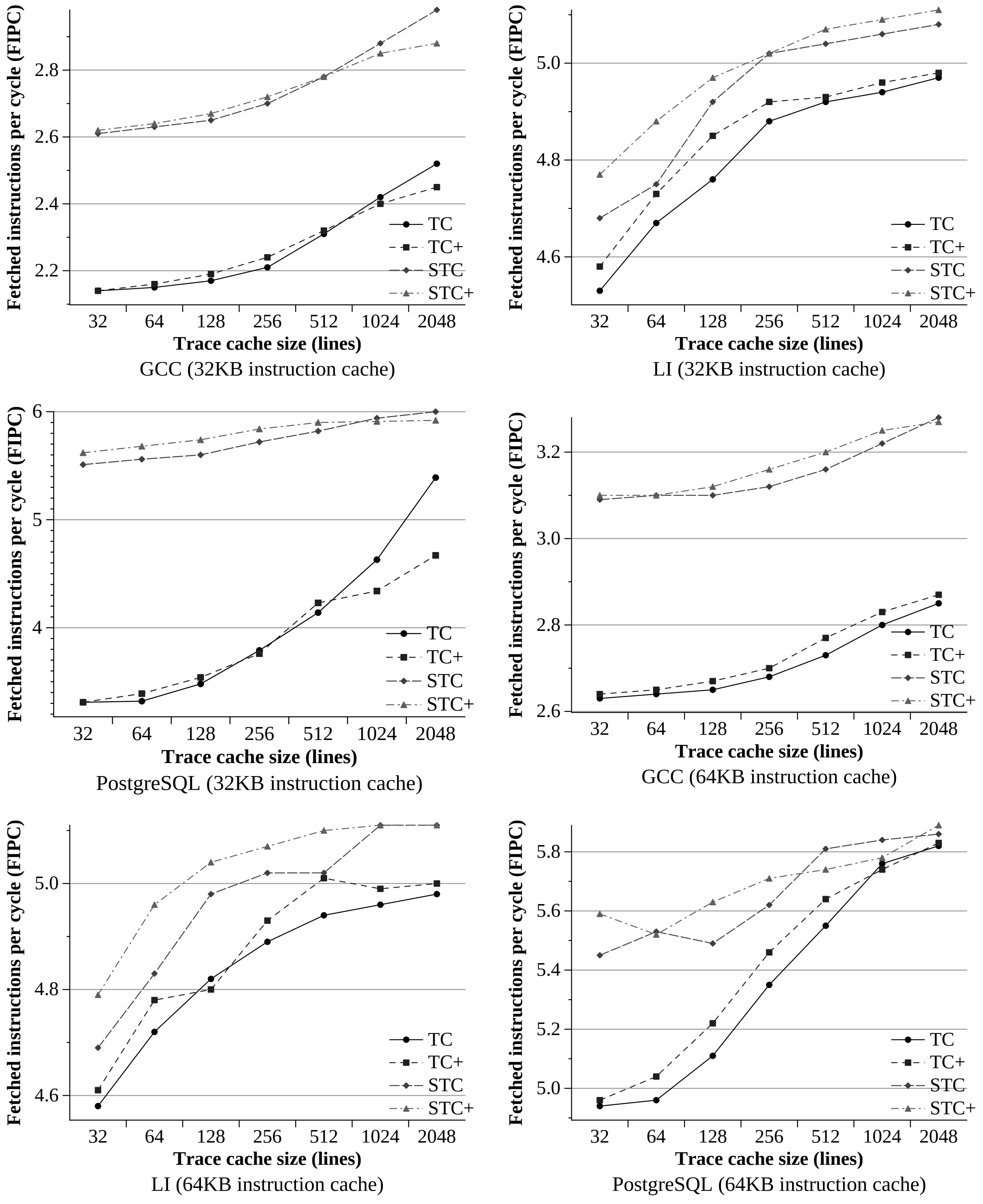


Figure 5.4. FIPC performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) and instruction cache sizes of 32 and 64KB using realistic branch prediction. The STS enhanced models are tagged with (+). Setups labeled STC use the optimized code layout.

5.4.2 Perfect branch prediction

Now, we examine the limits of STS under perfect branch prediction. Figure 5.5 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 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 5.5 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.

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 5.1 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.

	Trace Length				
Bench.	All	orig		STC	
	traces	Blue	Red	Blue	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 5.1. 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.

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.

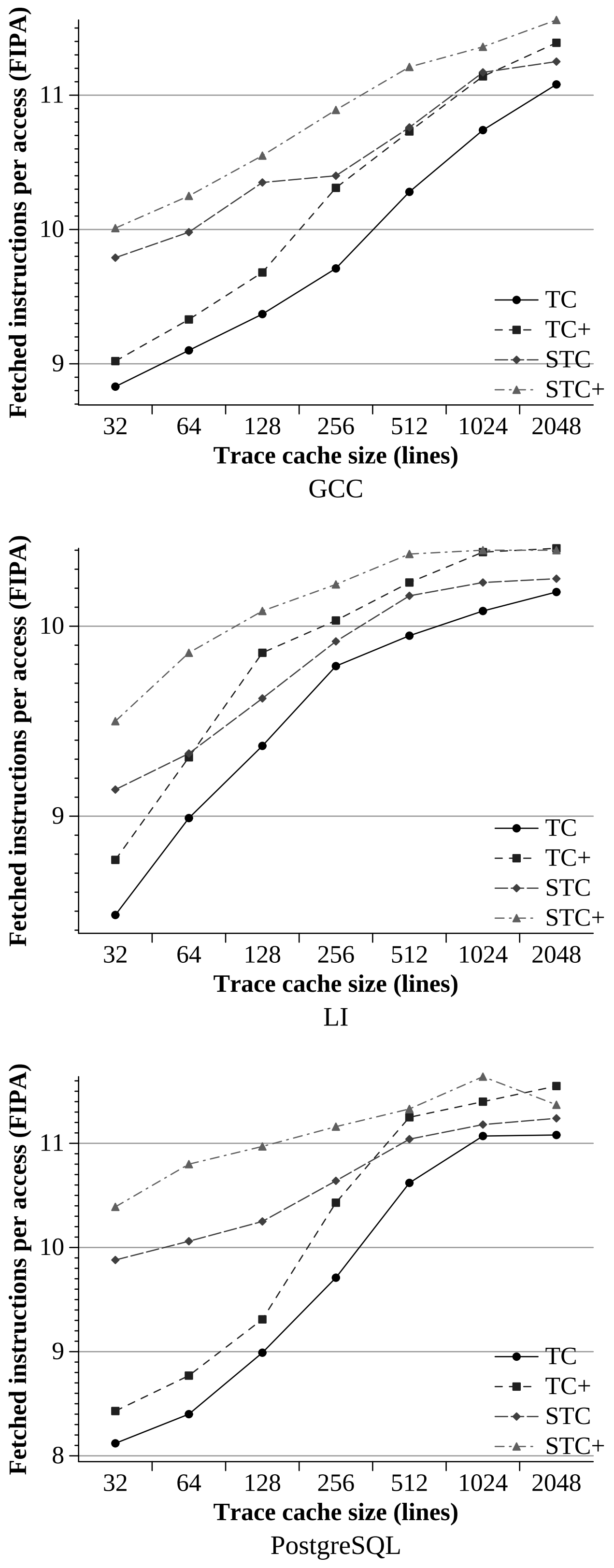


Figure 5.5. FIPA performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) using perfect branch prediction. The STS enhanced models are tagged with (+). Setups tagged STC use the optimized code layout.

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.

Fetched Instructions Per Cycle

Figure 5.6 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.

5.5 Conclusions

This chapter shows that there is a large redundancy between the software trace cache and the trace cache mechanism, as both are doing the same task: storing sequentially executed basic blocks in consecutive memory storage. The difference is that the STC does it at compile-time, while the trace cache does it at run-time.

The problem arises when the same traces are being constructed both by the compiler and the architecture. These traces are being stored both in the trace cache and the instruction cache, a redundant use of resources which does not contribute to performance, because it is possible to obtain the whole trace in a single cycle from either cache.

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 small 32-entry trace cache than a 2048-entry one without any of these improvements.

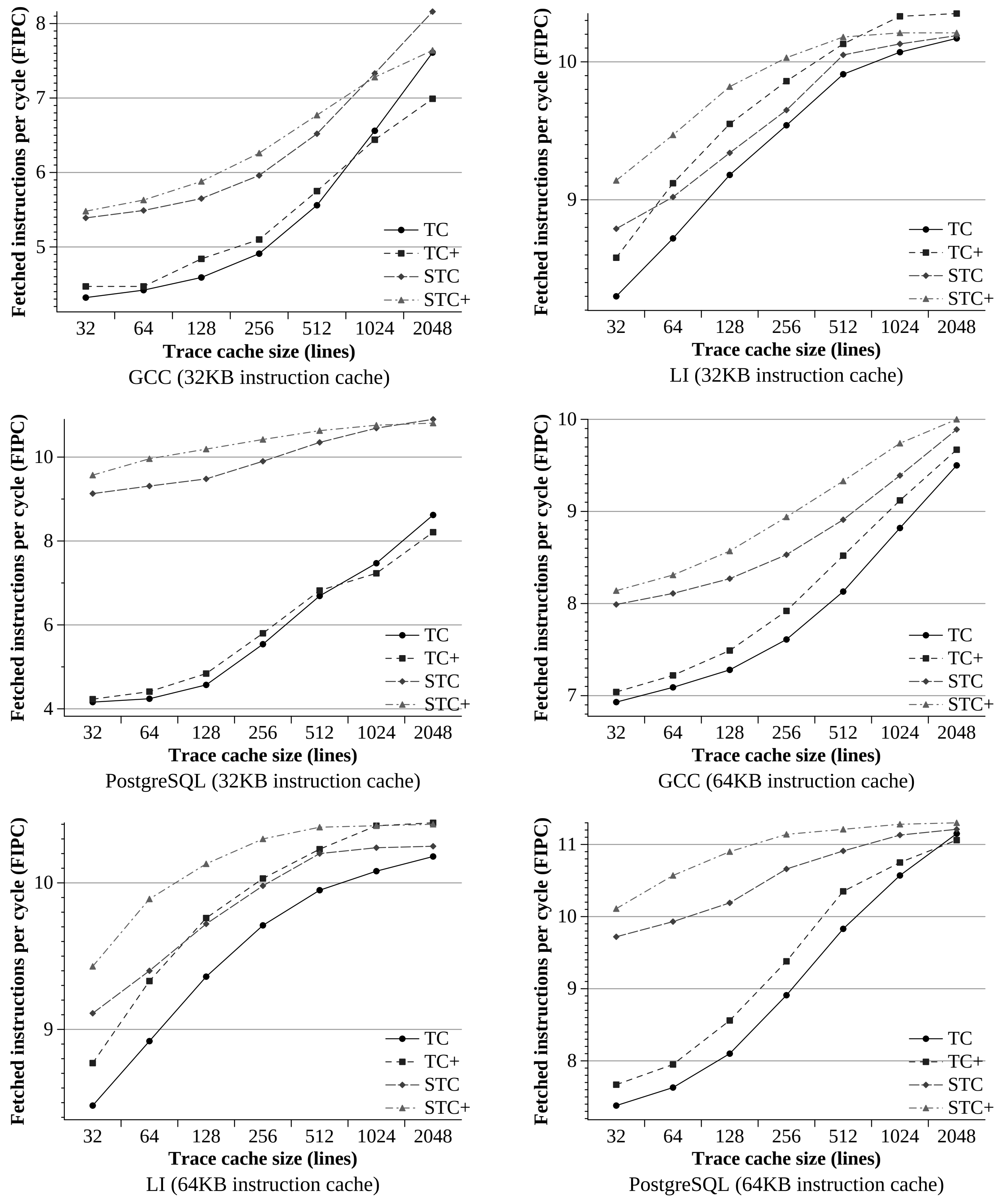


Figure 5.6. FIPC performance for trace cache sizes from 32 to 2048 entries (2KB to 128KB) and instruction cache sizes of 32 and 64KB using perfect branch prediction. The STS enhanced models are tagged with (+). Setups tagged STC use the optimized code layout.