# 4 SOFTWARE TRACE CACHE

Our first approach to high fetch performance is the use of compiler optimizations to optimize the layout of instructions in memory, so that the code makes a better use of the underlying hardware resources regardless of the specific details of the processor/architecture.

The Software Trace Cache (STC) is a code layout algorithm with a broader target than previous layout optimizations. We target not only an improvement in the instruction cache hit rate, but also an increase in the effective fetch width of the fetch engine.

The STC algorithm organizes basic blocks into chains trying to make sequentially executed basic blocks reside in consecutive memory positions, then maps the basic block chains in memory to minimize conflict misses in the important sections of the program.

We evaluate and analyze in detail the impact of the STC, and code layout optimizations in general, on the three main aspects of fetch performance: the instruction cache hit rate, the effective fetch width, and the branch prediction accuracy.

Our results show that layout optimized codes have some special characteristics that make them more amenable for high performance instruction fetch: they have a very high rate of not-taken branches, and execute long chains of sequential instructions; also, they make a very effective use of instruction cache lines, mapping only useful instructions which will execute close in time, increasing both spatial and temporal locality.

## 4.1 Placement algorithm

The Software Trace Cache (STC) layout algorithm is largely based on the work of Torrellas *et al.* [87], which in turn is based on the work of Hwu and Chang [32].

The STC presents several improvements on the previously proposed algorithms. As opposed to a manual seed selection, based on source code and profile data analysis, we use an automatic process for selecting the starting point of our basic block traces. Our basic block chaining algorithm does not use the ExecThreshold and BranchThreshold parameters used in [87]. Instead, we

build all our basic block traces in a single pass of the algorithm, without any user intervention to determine threshold values.

Finally, we map whole basic block traces into the Conflict Free area (CFA) instead of mapping individual basic blocks as done in [87]. Mapping whole traces may introduce less frequent basic blocks in the CFA, but allows us to exploit more spatial locality, which proves important to the effective fetch width.

### 4.1.1 Seed selection

Our 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.

Running the training set on each benchmark, we 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.

Before we can organize the basic block set into traces, we need to select the *seeds* or starting points for those traces. In [87], the operating system code is studied in detail to find the most frequent entry points, and so a few subroutines are selected (the page fault handling routine, ...). In [68] we analyze the code of a relational database management system (DBMS) and select the entry points for the different query operations as seeds.

However, a detailed analysis of source code is not always feasible nor desirable. For this reason we have selected *all* subroutine entry points as seeds. We maintain the list of seeds ordered by basic block weight: from the most frequently executed seed to the least executed one. We explore each seed in turn, ignoring those seeds which have already been included in a previous trace, recording the order in which traces were created.

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

### 4.1.2 Trace construction

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

For example, following the graph in Figure 4.1.a, the algorithm starts from seed A1. From basic block A1 the algorithm selects the most likely outgoing path, which leads to block A2. From basic block A2, the most likely outgoing path leads to an already explored seed C1. Discarded block B1 is already a seed, and will be explored later. The trace starting from seed C1, and containing blocks C1 to C4 (excluding block C5) is then inlined after block A2. The algorithm continues at the next sequential block A3 (the return point for trace C1-C4).

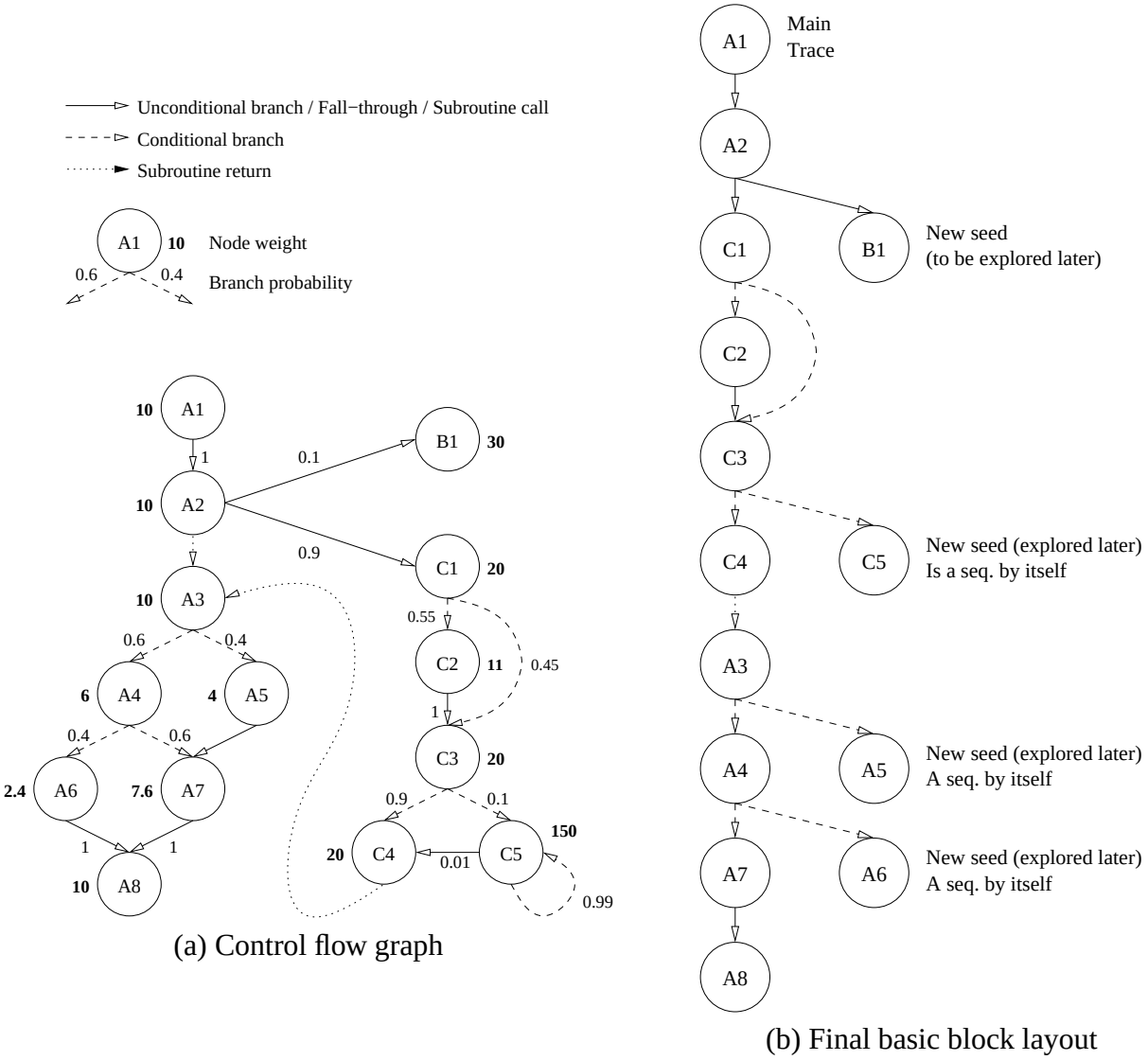



**Figure 4.1. Example of the Software Trace Cache basic block chaining algorithm. Basic blocks are mapped so that the execution trace is consecutive in memory.**

From basic block A3, the most likely outgoing path leads to block A4. Discarded block A5 is added to the list of unvisited seeds, which is maintained in weight order. From basic block A4, the algorithm visits blocks A7 and A8, adding discarded block A6 to the seed list. Figure 4.1 shows the resulting trace, including basic blocks from both routine A and routine C.

The chain inlining step is a novel contribution of the STC on top of what was done in [32, 87]. It allows the STC to build long basic block chains without need of a careful seed selection based on source code analysis, and makes the use of threshold values unnecessary.

### 4.1.3 Trace mapping

As shown in Figure 4.2, we map the resulting traces in the order they were created: from the most frequently executed one, to the least executed one. In this way, we map equally popular traces next to each other, reducing conflicts among them. Also, we divide traces in instruction cache sized chunks, and leave an empty space at the beginning of each block except the first one (the one containing the most popular traces).

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

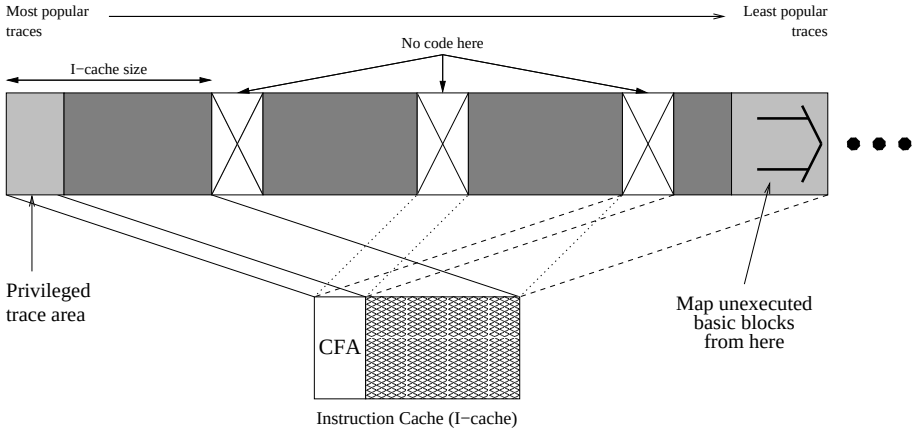



**Figure 4.2. Trace mapping for a direct mapped instruction cache.**

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

As a difference with previous work, we use heuristics in order to determine an adequate CFA size without requiring a trial and error approach. Figure 4.3 shows an example of how we determine an appropriate size for the CFA.

We take the most popular traces, one at a time. Then we compare the percentage of the total execution time that it gathers compared to the percent of the instruction cache that it requires. If the execution percent is higher than the space taken in the cache, we include the trace in the CFA. We then add the next trace, and consider the percent of the execution they take together, and the fraction of cache they require. As long as the fraction of execution is larger than the fraction of

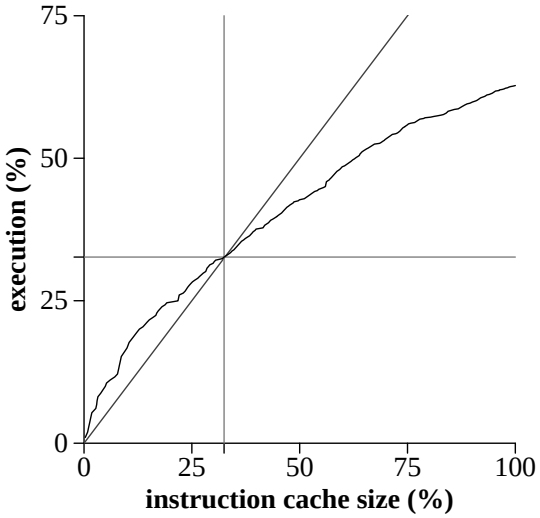



**Figure 4.3. Determining the size of the CFA from the execution frequency and size of the most popular traces.**

instruction cache they require, we keep adding traces to the CFA trying to balance the two factors. In the example of Figure 4.3 we would devote 32% of the instruction cache to the CFA, as it gathers exactly 32% of the program execution.

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

## 4.2 Performance impact

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

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

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

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

Finally, we analyze the impact of layout optimizations in other elements beyond the fetch engine, and fins that they not only have a positive impact on the instruction memory hierarchy, but

that they also improve *data* memory performance, due to a reduced interference between instructions and data.

### 4.2.1 Impact on the instruction cache

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

Figure 4.4 shows the instruction cache miss rate of a baseline cache setup compared to that of the same cache running optimized codes, and two hardware optimized setups. The code layout optimizations explored are those proposed by Pettis & Hansen (PH) [59], Torrellas *et al.* (TXD) [87], and the Software Trace Cache (STC). The hardware optimized setups are a 2-way set associative cache, and a 16-way fully associative victim buffer. None of the hardware optimized setups uses an optimized code layout.

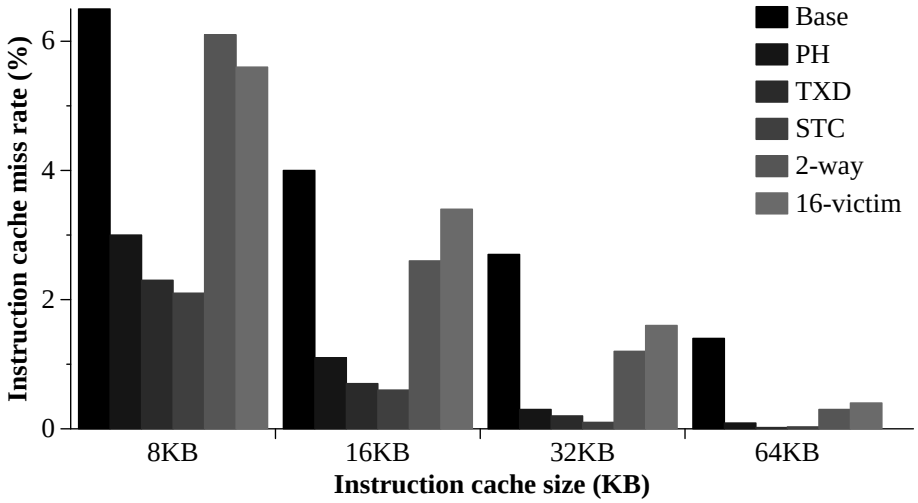



**Figure 4.4. Instruction cache miss rate for various cache sizes when using different hardware configurations and code layout optimizations.**

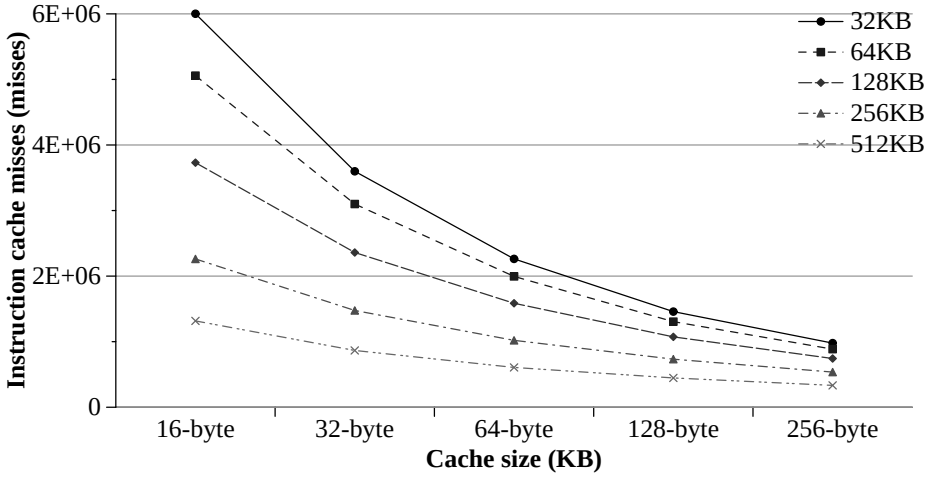
The results in Figure 4.4 show that code layout optimizations have a very significant impact on the instruction cache miss rate for all explored cache sizes, much larger than the two hardware optimizations explored. The instruction cache miss rate of a 16KB instruction cache running optimized codes is lower than that of a 64KB cache running unoptimized codes. This shows that optimized codes make a more effective use of the available cache space, requiring a smaller cache to fit the instruction working set.

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

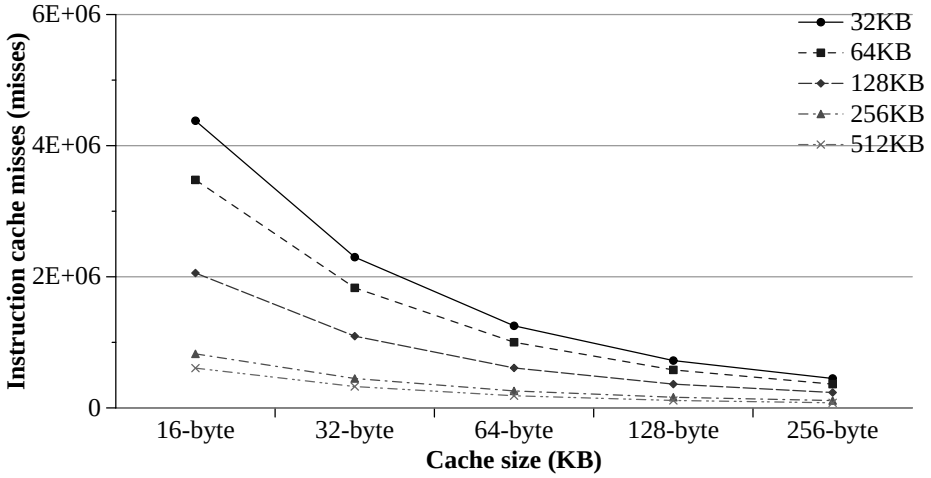
Code layout optimizations are very effective at reducing instruction cache miss rates. The usual explanation for this miss rate reduction is that a careful layout of the routines may reduce the number of conflict misses, and that is the main aspect where code layout optimizations differ

from each other. However, we will show that layout optimizations not only have an impact on conflict misses.

Figure 4.5 shows the number of instruction cache misses of two version of a commercial database management system (DBMS) running an OLTP workload (TPC-B). Commercial databases are very large codes, with flat execution profiles, which suffer from heavy capacity problems rather than conflict misses.




(a) Baseline DBMS application




(b) Optimized DBMS application

**Figure 4.5. Instruction cache misses for various cache and line sizes for a commercial database management system running an OLTP workload.**

The results in Figure 4.5 show that code layout optimizations also have a significant impact on the number of misses of such big workloads, although the number of conflict misses can not be reduced because the working set is too large to fit in the cache regardless of the layout of routines.

Figure 4.6 shows the relative number of misses of the optimized DBMS application compared to the unoptimized code. That is, for each instruction cache and line size, the graph shows the percentage of misses still present in the optimized application. For example, on a 64KB cache with 128-byte lines, the optimized binary has only 45% of the misses of the unoptimized code.

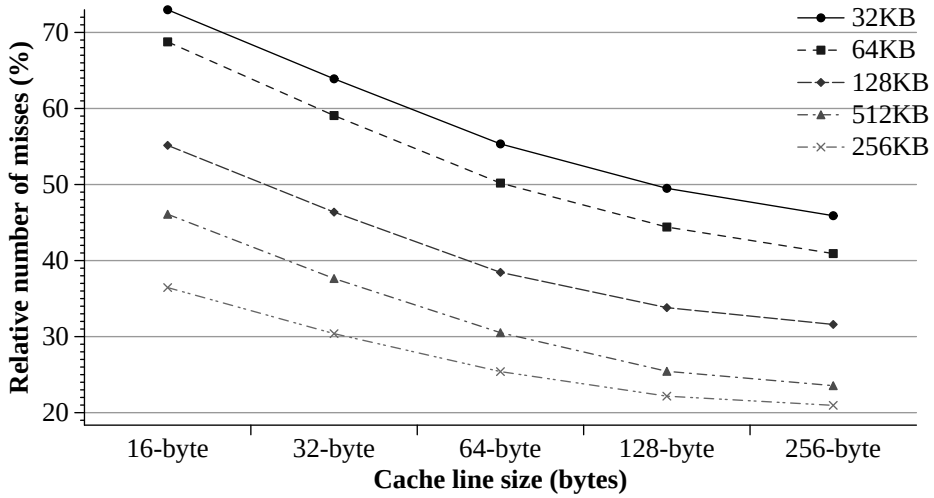



**Figure 4.6. Relative number of misses in the optimized DBMS binary compared to the baseline application for various cache and line sizes.**

The results in Figure 4.6 show that even for large workloads which do not fit in the instruction cache, code layout optimizations can obtain important miss reductions (up to an 80% reduction for a 256KB cache with 256-byte lines).

Further analysis of these results show that larger caches obtain better miss reductions. This trend holds up to the 256KB cache, because the workload already fits in a 512KB cache. The same trend is present for the instruction cache line size: longer cache lines obtain better miss reductions. The results if Figure 4.5 show that the unoptimized application does improve performance as the cache line and size increase, but the optimized application improves faster than the baseline.

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

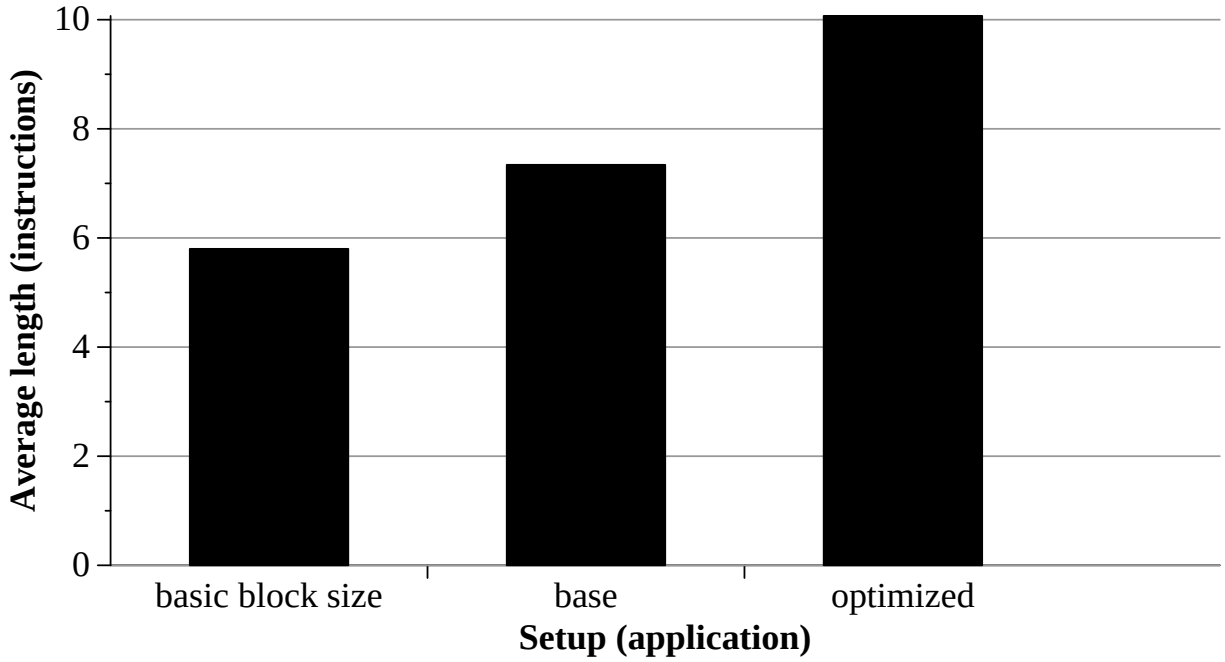
### Spatial locality

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

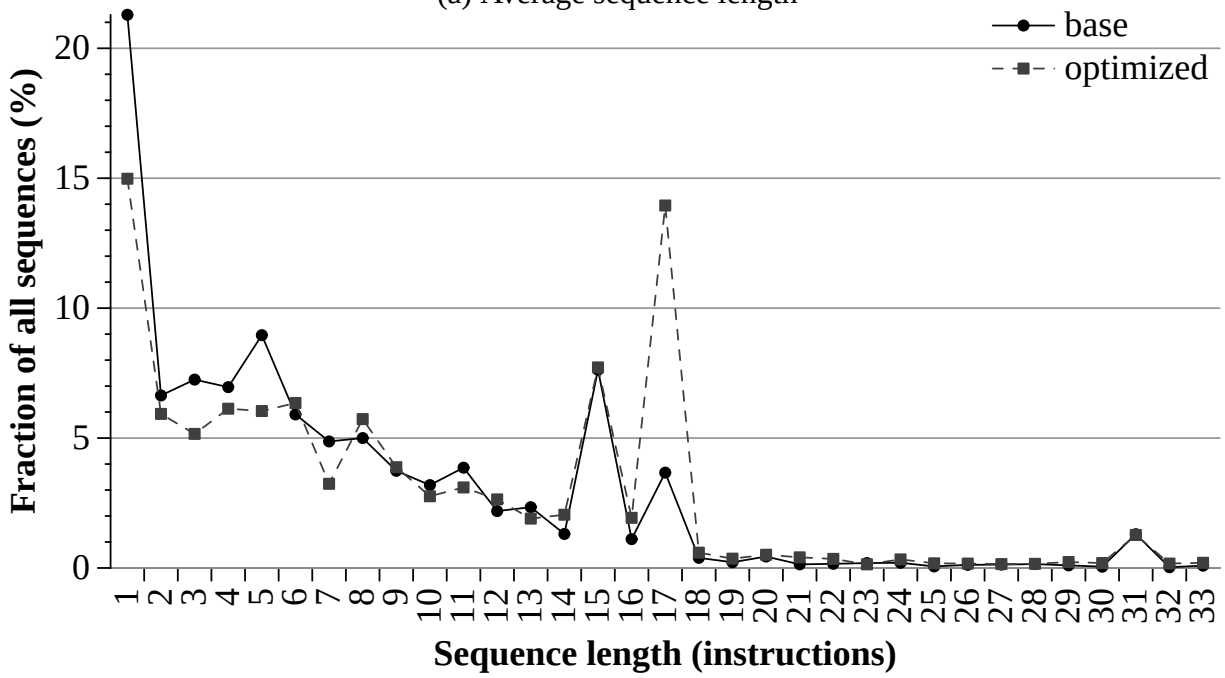
Figure 4.7.a shows the sequence length increase experienced by our commercial database engine. We compare the basic block size with the average number of sequentially executed instructions in the baseline (unoptimized) application, and the optimized application.

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

Figure 4.7.b shows a detailed breakdown of the number of sequences of each length for both binaries. The graph shows that there is a 30% decrease in the number of sequences of length 1, and a large increase in the number of sequences of length 17. That is, we are reducing the number of short sequences, and increasing the number of long sequences. However, there is still more spatial locality than that explained by the basic block chaining optimization.




(a) Average sequence length




(b) Breakdown of the different sequence lengths

**Figure 4.7. Code layout optimizations increase the number of sequentially executed instructions.**

Figure 4.8 shows the percentage of times that a number of unique words is used in a 128-byte cache line before it is replaced (32 instructions per cache line), for both the baseline and optimized application.

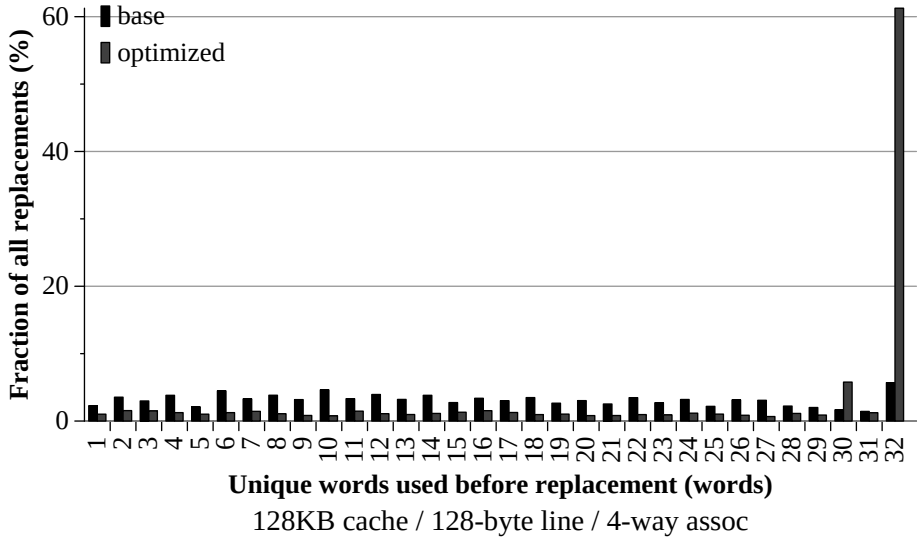



**Figure 4.8. Layout optimized codes use all the instructions in a cache line before it is replaced.**

The results in Figure 4.8 show that the optimized application uses the whole cache line over 60% of the time. That is, in most cases, all instructions in a cache line will be executed at least once before that cache line is replaced. Such behavior is not present in the unoptimized application, and would explain the improved instruction cache performance.

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

The routine splitting optimization separates the useful instructions from those which will rarely or never be executed, which reduces the size of the procedure. Then the procedure ordering moves the useless instructions away, and maps procedures which execute close in time next to each other. After this optimizations, not only we execute longer sequences of instructions, but when a sequence is terminated, it is likely that the target sequence is in the same cache line.

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

### Temporal locality

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

Figure 4.9 shows the number of cycles that a given line has been present in the cache before being replaced. That is, we measure the lifetime of a cache line, from the moment it is loaded into

the cache to the moment it is evicted. Note that the X axis showing the lifetime is in a logarithmic scale: a single step through the axis means the cache line was active for double the amount of time.

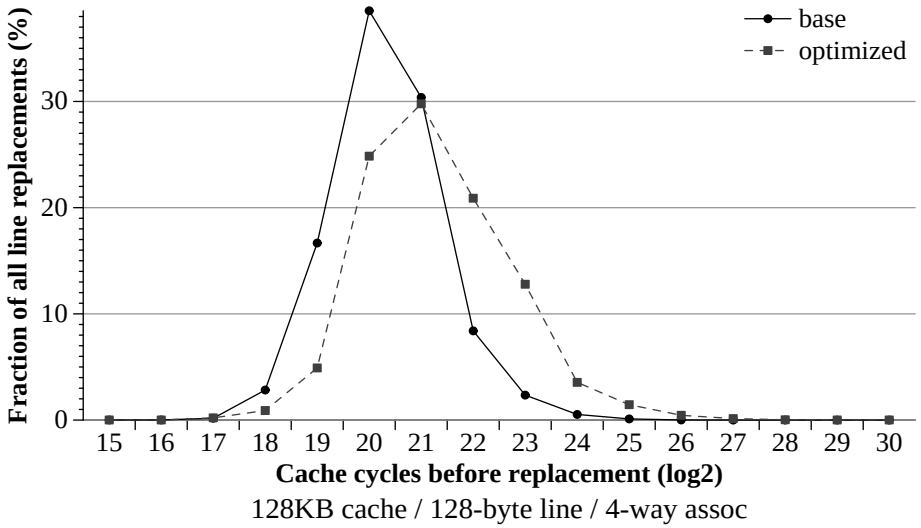



**Figure 4.9. Instruction cache lines have an increased lifetime in layout optimized codes.**

Our results show that cache lines have an extended lifetime in the optimized binary. The average lifetime has moved from $2^{19}$ cycles to $2^{20}$ or more cycles, meaning that cache lines are available for twice the amount of cycles. Because we require fewer cache lines, we can keep a given cache line for longer before having to replace it, offering more opportunities for temporal reuse of instructions.

Figure 4.10 shows the average number of times that a given instruction is used every time it is loaded into the cache. That is, every time we load a cache line, we count how may times each instruction was used before the line was replaced.

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

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

### 4.2.2 Impact on the fetch width

The layout of basic blocks in memory may also have an effect on the effective fetch width. Figure 4.11 shows an example of how an optimized code increases the number of instruction that can be fetched per cycle.

The presence of branches disrupts the fetch sequence, but it is taken branches which actually interrupt it. It is difficult to fetch both a taken branch and its target in the same cycle, as is done in the branch address cache [94] and the collapsing buffer [14]. It requires fetching multiple cache lines per cycle, and a complex instruction alignment network which may add extra pipeline stages.

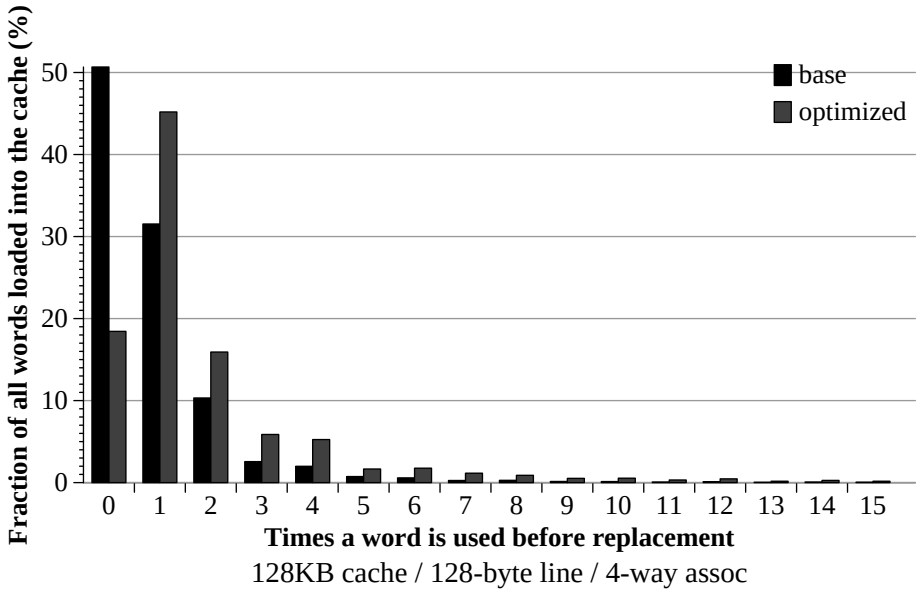



**Figure 4.10. Almost all instructions loaded in the cache are used at least once, and instruction reuse increases in optimized codes.**

(a) It is hard to fetch a branch and its target in the same cycle

(b) It is easy to fetch past a not taken branch

**Figure 4.11. Code layout optimization increase the fetch width by aligning branches towards not taken.**

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

As will be shown in Section 4.2.3, code layout optimizations are very successful at aligning branches towards their not taken direction, reaching an 80% not-taken rate among conditional branches. Furthermore, 60% of all executed branches are always not taken.

Figure 4.12 shows the impact of code layout optimizations on a fetch engine capable of fetching up to 3 sequential basic blocks per cycle (the SEQ.3 engine describes in [74]), and a trace cache architecture.

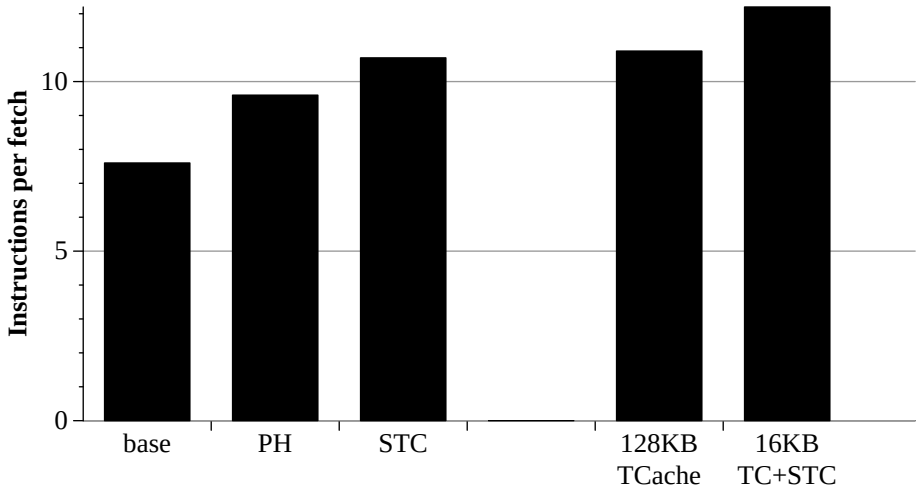



**Figure 4.12. Code layout optimizations effectively increase the fetch width of baseline and trace cache fetch architectures.**

The results in Figure 4.12 show that code layout optimizations such as the one proposed by Pettis & Hansen [59] and the Software Trace Cache effectively increase the number of instructions provided by the fetch engine each cycle, reaching a performance level close to that of a trace cache.

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

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

The trace cache reads the dynamic instruction stream, and so is unaffected by the layout of instructions in memory. However, the trace cache is not a stand-alone fetch mechanism. If the requested trace is not present in the trace cache, it has to be fetched from a secondary fetch path, usually a sequential fetch engine. It is in those cases when code layout optimizations help a small trace cache to increase performance: if the secondary fetch engine has a performance close to that of the trace cache, it is less critical to miss in the trace cache.

### 4.2.3 Impact on the branch predictor

We have shown that code layout optimizations have a positive impact on the instruction cache performance, and that they increase the effective fetch width, but we have not examined the impact of the code layout on the branch prediction mechanism.

A better instruction cache performance means that instructions can be provided faster, without waiting for the lower memory hierarchy levels. An increased fetch width means that each time we fetch instructions, a larger amount of instructions is provided. But if we impact negatively the branch prediction accuracy, we will be fetching very fast, and very wide, but from a wrong speculative path.

#### Effect on static prediction

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

Figure 4.13 shows the branch prediction accuracy of some simple static branch prediction strategies (always taken, always not taken, backward taken forward not taken) and the profile based predictor for both the original code layout and the compiler optimized layouts. For the optimized layout, we show results for the same input set used for training (self-optimized) and for a different input set (cross-optimized). The prediction accuracy of an 8KB Gshare predictor is shown for comparison purposes.

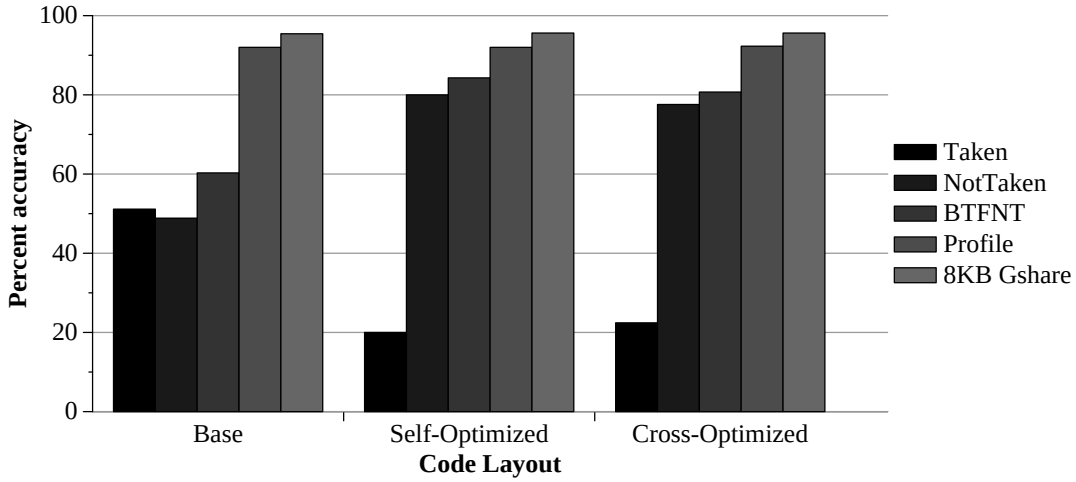



**Figure 4.13. Static branch prediction accuracy for the original and optimized code layouts (self and cross trained).**

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

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

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

To gain further insight on this high predictability of optimized binaries, we explore in depth the changes in branch behavior introduced by the code layout optimization. Figure 4.14.a shows a classification of all dynamic branches by the percentage of times they are taken or not taken for both the original and the optimized code layouts. Branches to the left of the plot are always not taken, while branches to the right are always taken.

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

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

The classification for the optimized code layout shows that we were quite successful at reversing the branch direction for those usually taken branches. The fraction of always taken branches is reduced from 32% to 10%, and most categories over 50% taken also present reductions in the number of branches. This leads to a significant increase in the number of always not taken branches, from 36% to 59%. With most highly biased branches in the not taken side, and most other branches moving from over 50% taken to mostly not taken, the prediction accuracy of an always not taken (or BTFNT) predictor, increases significantly, as we have seen in Figure 4.13.

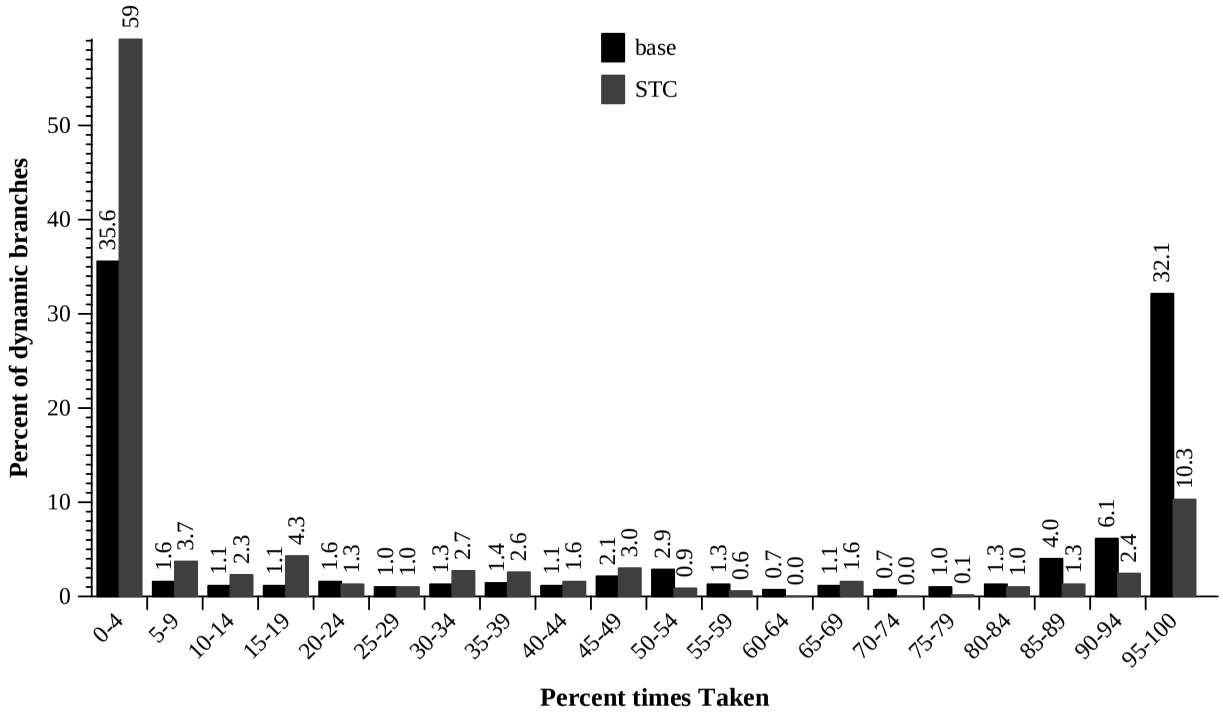
As shown in Figure 4.14.b the average number of branches in the 0-4% taken class does not actually represent the typical behavior of our benchmarks. Only two benchmarks stand above the average: *vortex* and *postgres* (with *perl* joining the group for the optimized layout), while all others stay well under the 59% average.

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

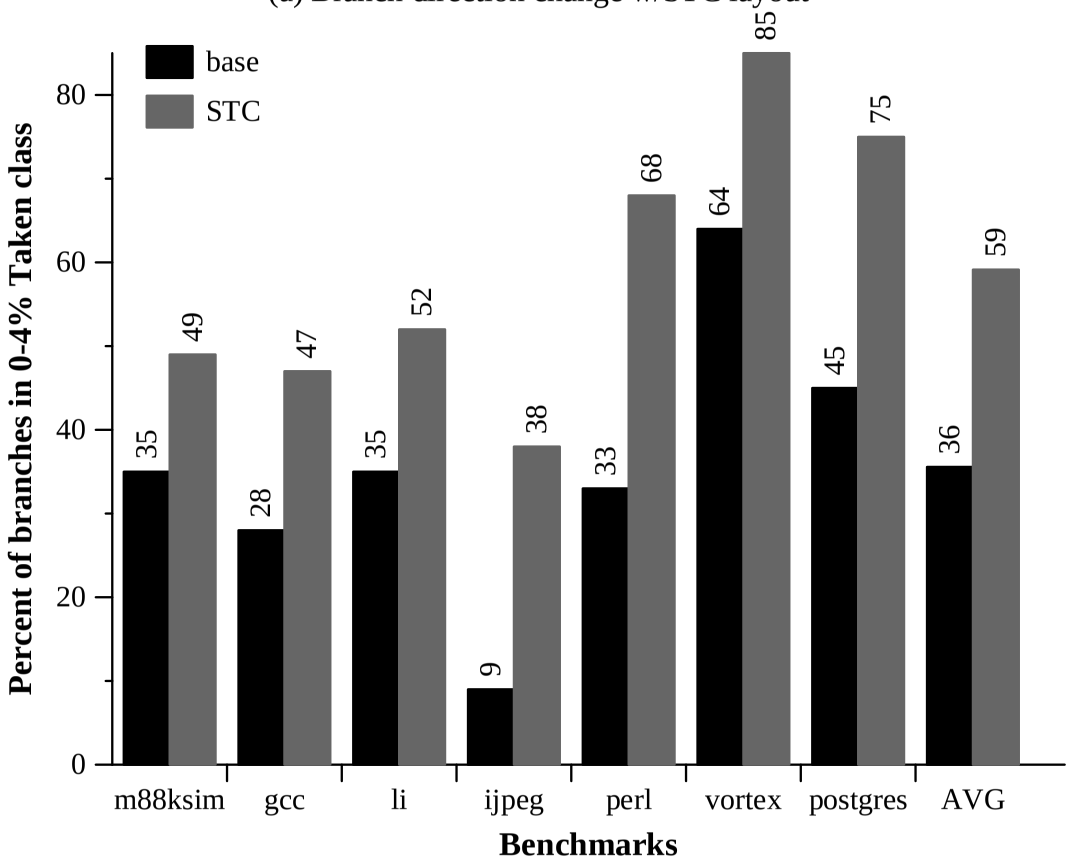
Next, we will examine how this change in branch direction affects dynamic branch prediction.

### Effect on two-level adaptive predictors

Figure 4.15 shows the effect of code reordering on dynamic prediction accuracy for the Gshare, PAg, and bimodal predictors. Predictor sizes from 512 bytes to 16KB are explored for both the baseline (dotted line) and the optimized code layout (solid line).




(a) Branch direction change w/STC layout




(b) Fraction of always not taken branches

**Figure 4.14. The use of optimized code layouts reverses branch direction, so that they tend to be usually not taken.**

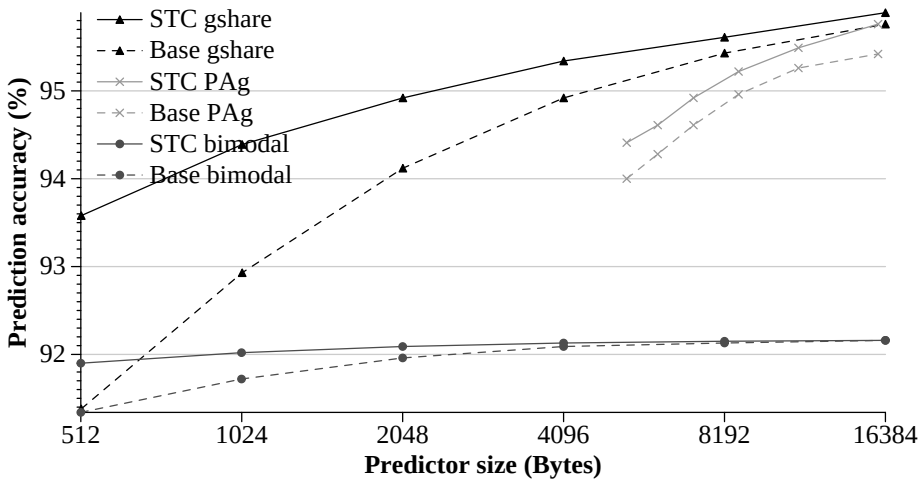



**Figure 4.15. Dynamic prediction accuracy for both the base and the STC optimized code layouts using two-level adaptive prediction schemes.**

Clearly, the STC increases the prediction accuracy of the examined branch predictors, specially for the smaller predictor sizes. Both the Gshare and the bimodal predictors seem to converge at infinite predictor size, which points that the benefits of using the STC are related to prediction table interference. The larger the table, the less interference, the closer the prediction accuracy for both layouts.

Figure 4.16 shows the percent of dynamic branches which introduce conflicts in the prediction tables of the gshare branch predictor with both the baseline and the optimized code layouts. We classify conflicts in three groups: neutral interference when the conflict does not change the prediction, and positive or negative if the conflict changes the prediction for good or bad.

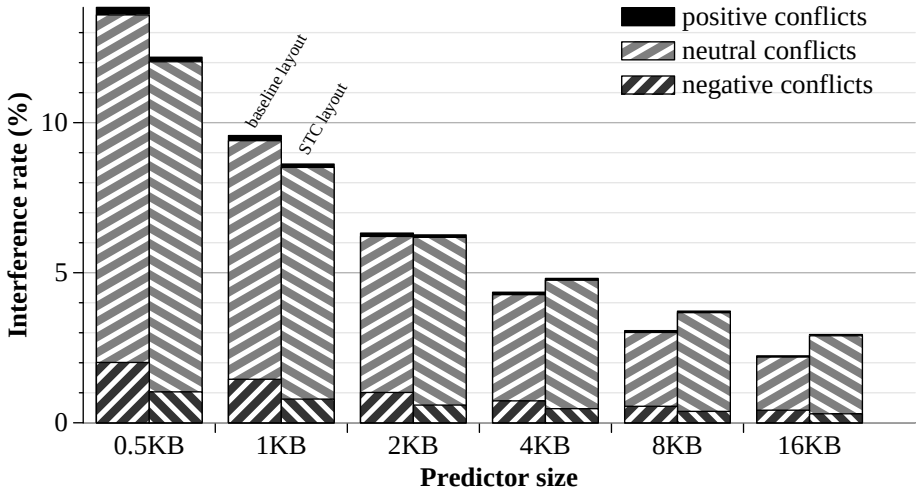



**Figure 4.16. Percent of dynamic branches which cause interference in the gshare prediction tables for the baseline and optimized code layouts.**

As expected, there is a significant reduction in the number of negative conflicts when the STC

layout is used with the Gshare branch predictor. For example, a 1KB gshare goes down from 1.45% of negative conflicts to 0.79% using the optimized code layout.

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

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

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

### Effect on dealiased predictors

Given that the use of an optimized code layout is reducing the negative interference found in the dynamic prediction tables, it is interesting to examine what happens with modern branch predictors that are already organized to minimize such interference like the agree [82], bimode [39], and gskew [46, 77] predictors. We will refer to these predictors as dealiased branch prediction schemes.

Figure 4.17 shows the prediction accuracy of the dealiased predictors with both the baseline and the optimized code layouts. The prediction accuracy of the gshare predictor with the optimized layout is shown for reference purposes.

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

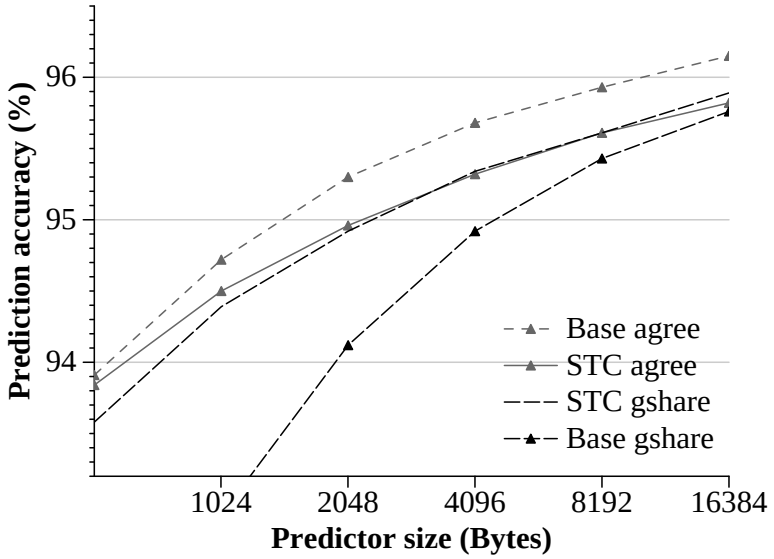
For medium and large predictor sizes, all dealiased branch predictors obtain higher accuracy with the baseline code layout, being the difference specially significant with the 16KB agree predictor, which obtains a 96.2% accuracy with the baseline layout and a 95.8% with the optimized code.

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

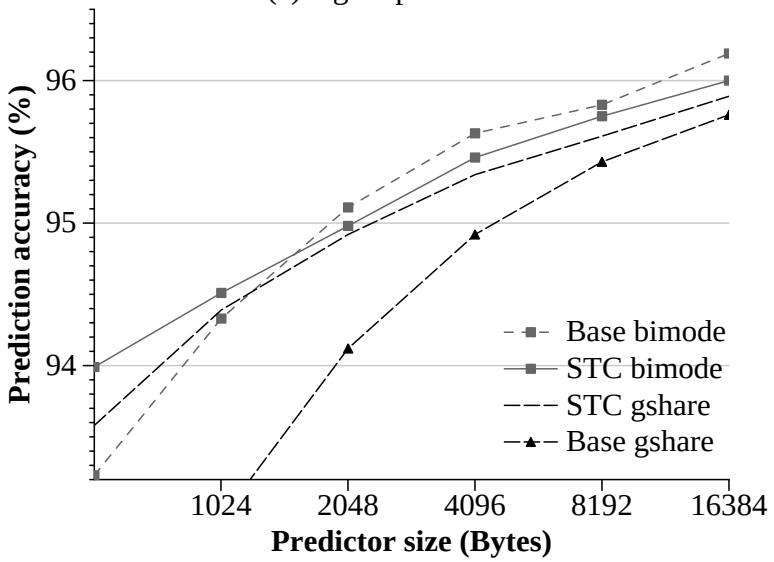
Figure 4.18 shows the percent of dynamic branches which introduce conflicts in the prediction tables of the gshare branch predictor with the optimized code layout and the agree predictor using both code layouts.

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

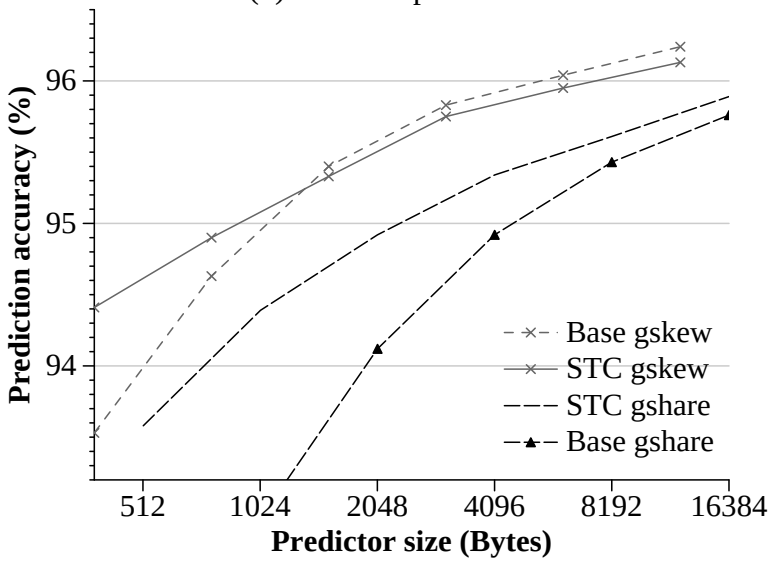
From these results it seems that the dealiased predictors prove more effective at reducing interference than the optimized code layout, but the more important result is that it seems more difficult to reduce conflicts in an optimized binary. The fact that the optimized code layout has more total interference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




(a) Agree predictor




(b) Bi-mode predictor




(c) Gskew predictor

**Figure 4.17. Effect of the optimized code layout on dealiased branch predictors.**

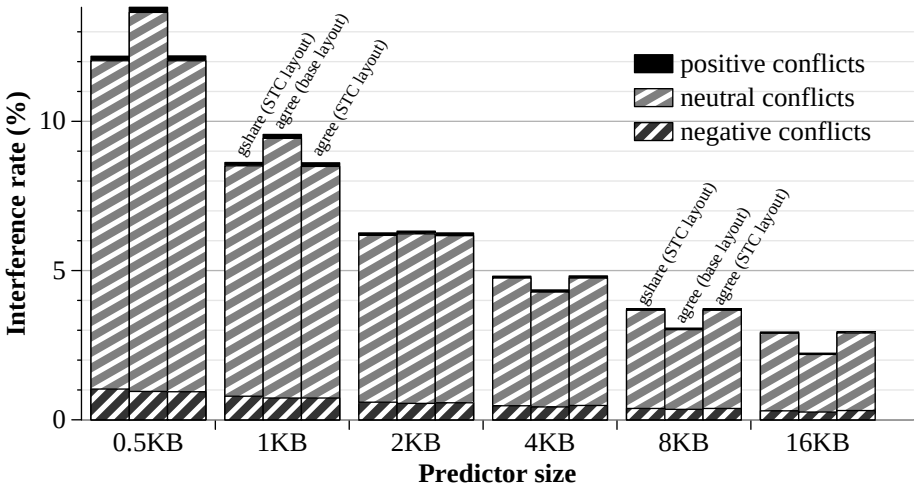



**Figure 4.18. Percent of dynamic branches which cause interference in the gshare prediction tables optimized code layout and the agree predictor using both code layouts.**

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

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

If that is the case, a branch prediction scheme solely based on a global branch history such as the GAg predictor should suffer heavily from this effect. Figure 4.19 shows the prediction accuracy of the GAg predictor using both code layouts, compared to the prediction accuracy of a gshare predictor.

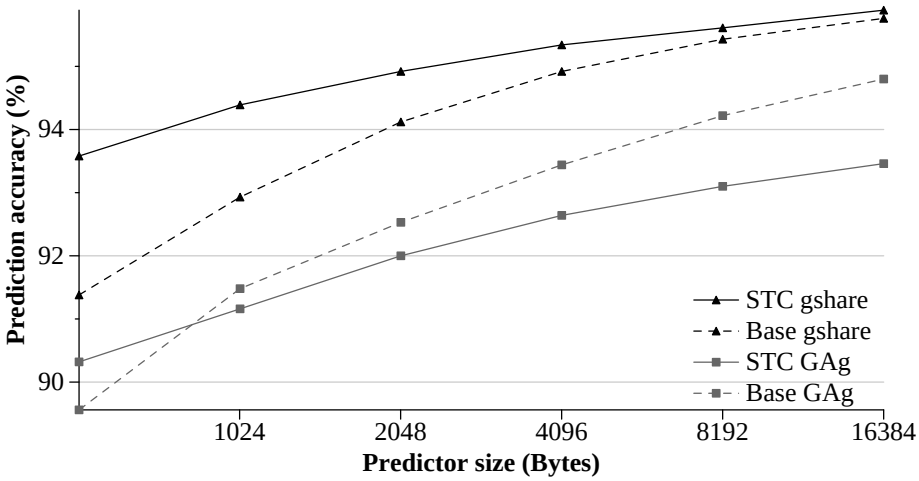



**Figure 4.19. Prediction accuracy of the GAg branch predictor compared to that of the gshare predictor using the baseline and the optimized code layouts.**

Clearly, the GAg predictor only benefits from the interference reduction obtained with the optimized layout for the smaller predictor size. For all other configurations the baseline layout obtains higher accuracy than the optimized one, confirming that the branch history distribution is causing a prediction accuracy loss. The gshare predictor XORs the branch address with the branch history, hiding this effect, which causes the interference reduction effect to dominate, increasing accuracy.

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

To analyze this BHR distribution factor, Figure 4.20 shows the number of times each possible history value was found in an 11-bit global history predictor for both code layouts. The BHR values are sorted by the number of zeros their binary value contains (from all 1's to all 0's). In addition to the BHR value usage, the figure shows the average usage, and the average + standard deviation. The average usage is the same in both code layouts. Note the Y axis is in $log_{10}$ scale.

The first remarkable aspect of these plots is the position of the highest peak. The most popular history value for the baseline layout is a BHR full of 1's (leftmost value), while the highest peak of the STC layout corresponds to a BHR full of 0's (rightmost value). Aside from that, the BHR value usage in the baseline layout is mostly spread across 1–2 orders of magnitude. Meanwhile, the STC layout has its BHR value usage spread across 4–5 orders of magnitude, with very high peaks on a reduced set of values. It is clear that values having mostly 1's are less used than those having mostly 0's.

To summarize these observations, we can just look at the distance between the average usage and the standard deviation lines. The more distance between them, the worse the BHR value distribution. In this case, the distance between both lines in the STC layout is 2.5x larger than in the baseline code layout.

### 4.2.4 Overall performance impact

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

Figure 4.21 shows the number of misses in the instruction TLB, and the shared L2 cache for a commercial database management system running an OLTP benchmark, using both unoptimized and optimized code. The misses in the shared L2 cache have been classified as either instruction misses, or data misses.

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

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

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

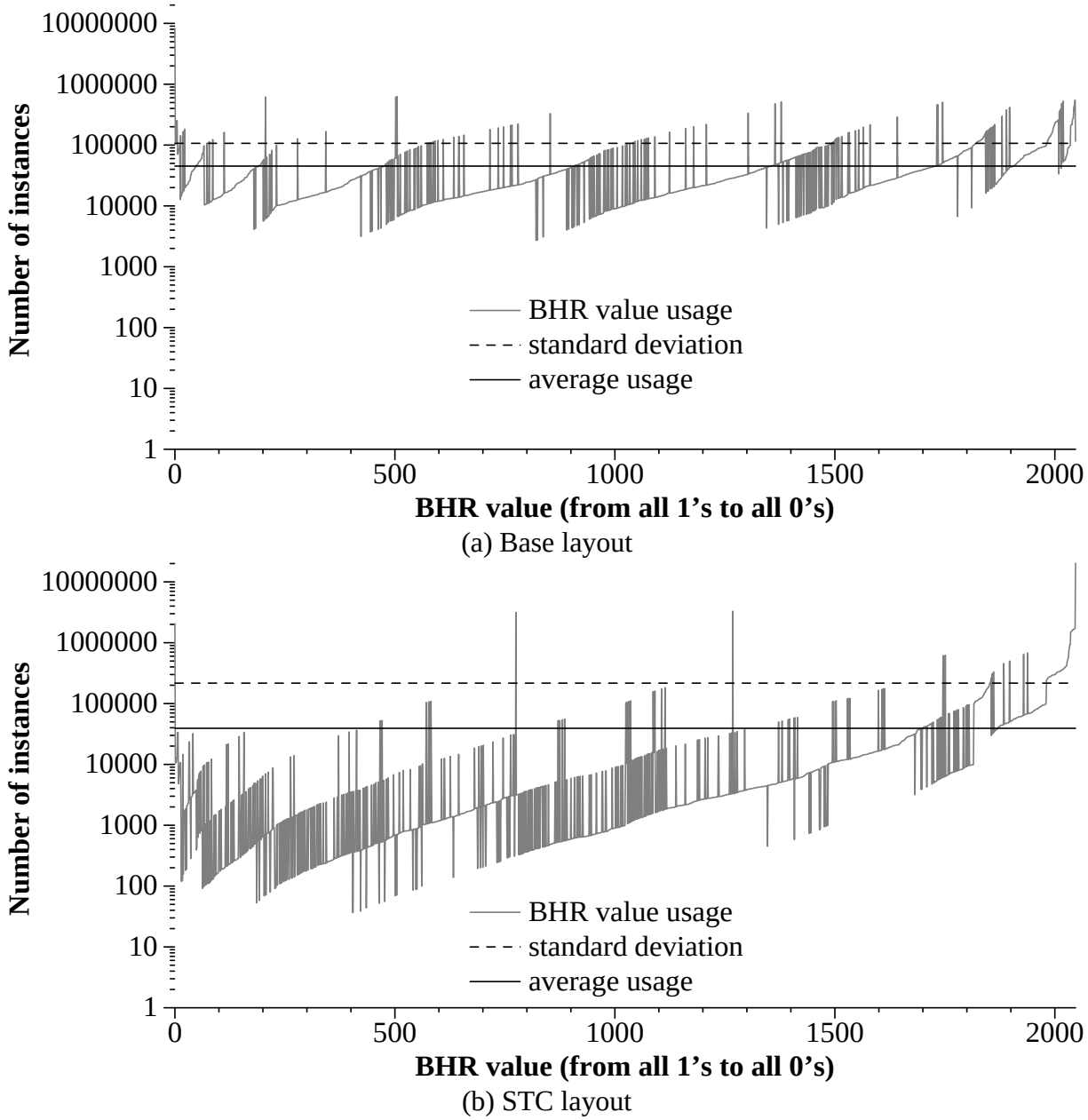



**Figure 4.20. Branch history register value distribution for the baseline code layout (a), and the STC optimized layout (b).**

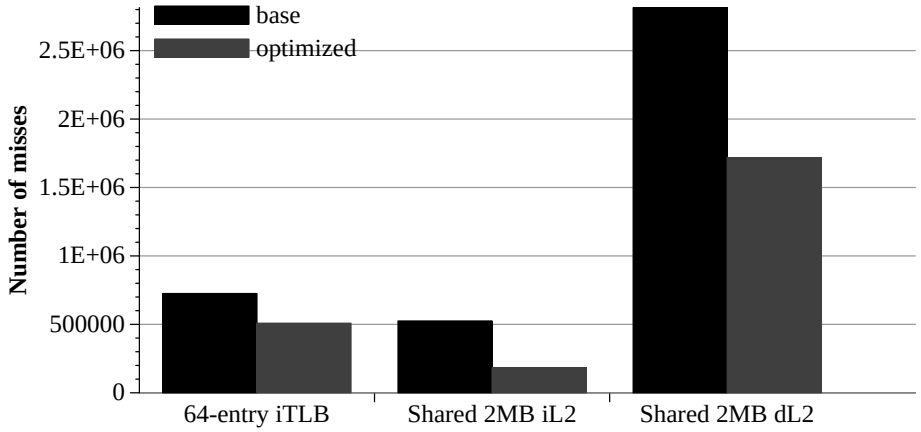



**Figure 4.21. Code layout optimizations impact not only the L1 instruction cache, but the whole memory hierarchy.**

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

Figure 4.22 shows the average processor performance measured in instructions per cycle (IPC) for the SPECint95 benchmarks using unoptimized and optimized codes for a variety of instruction cache sizes, and a perfect instruction cache. Results are shown for a processor with a realistic branch predictor, and a perfect branch predictor.

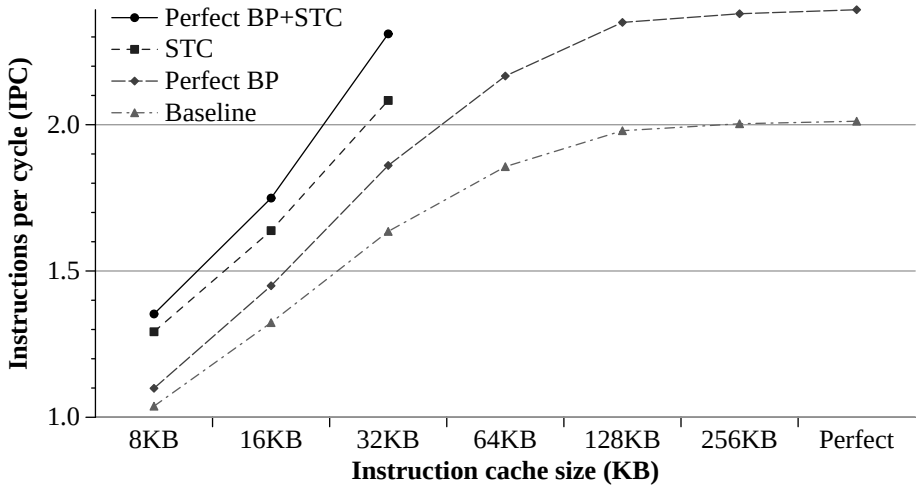



**Figure 4.22. Overall processor performance increases beyond the perfect instruction cache using code layout optimizations.**

The results in Figure 4.22 show that processor performance using layout optimized codes is higher than that of unoptimized codes using an instruction cache of twice the size. Moreover, the performance of the unoptimized binaries saturates after 128KB are devoted to the instruction cache, while the performance of optimized codes with a 32KB cache is higher than that of unoptimized codes using a perfect instruction cache.

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

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

Figure 4.23 shows the relative execution time of our commercial database application as we include different code layout optimizations. The optimization combinations explored include: procedure ordering alone (porder), basic block chaining alone (chain), basic block chaining with procedure splitting (chain+split), basic block chaining with procedure ordering (chain+porder), and all optimizations together (chaining, splitting, and ordering). We show results for real machine runs on two different Alpha platforms.

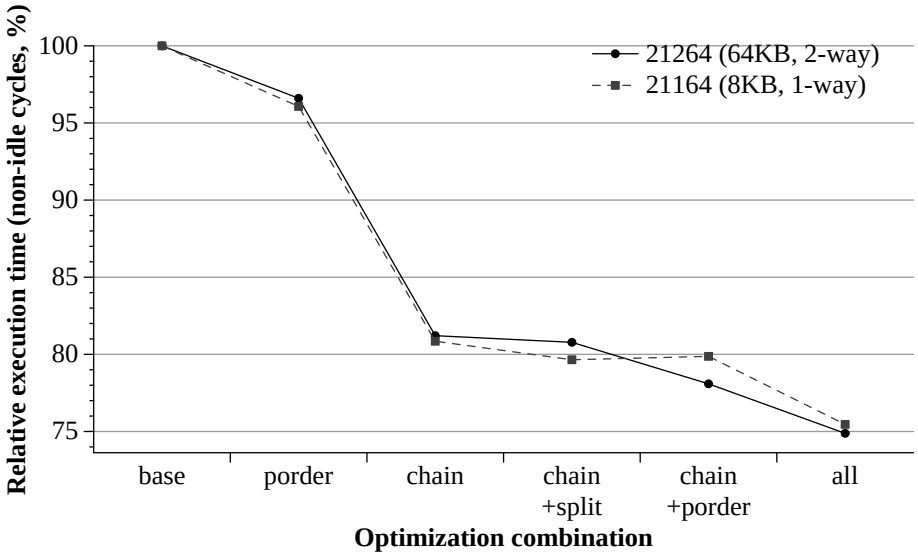



**Figure 4.23. Impact of several code layout optimizations on the overall system performance.**

Our results show that most of the performance improvement obtained derives from the basic block chaining optimization, which is the main responsible for the increased spatial locality experienced.

The next big step in performance is encountered when we add routine splitting and procedure ordering on top of the basic block chaining. The routine splitting provides an extra degree of freedom to the procedure ordering optimization, which now can move away the unused portions of a routine, compacting the code so that most cache lines contain only useful instructions.

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

## 4.3 Conclusions

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

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

Our results show that code layout optimizations are very effective at improving the instruction cache performance. A detailed analysis of the reasons for this improvement shows that in addition to a reduction in the number of conflict misses, optimized codes make a much more effective use of the available cache space, packing only useful instructions in a cache line, and moving unused sections of the code towards the end of the executable. This tight packing of instructions leads to a high increase in spatial locality, and an increased lifetime of cache lines, which offers extended opportunities for temporal reuse.

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

Having a positive impact on the instruction cache and the fetch width may be worthless if we are decreasing the branch prediction accuracy. But we show that such is not the case. Layout optimized codes are more amenable to branch prediction using either static branch predictors or simple 2-level adaptive branch predictors. Only for dealiased branch predictors, which use hardware mechanisms to remove branch aliasing from their prediction tables we experience a slight performance drop in the branch predictor when using optimized codes. However, the loss in prediction accuracy is more than compensated by the increased cache hit rate and fetch width.

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

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