Techniques for Enlarging Instruction Streams

Oliverio J. Santana, Alex Ramirez, and Mateo Valero
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
email:{*osantana,aramirez,mateo*}@ac.upc.edu

Abstract

This work presents several techniques for enlarging instruction streams. We call stream to a sequence of instructions from the target of a taken branch to the next taken branch, potentially containing multiple basic blocks. The long size of instruction streams makes it possible for a fetch engine based on streams to provide high fetch bandwidth, which leads to obtaining performance results comparable to a trace cache. The long size of streams also enables the next stream predictor to tolerate the prediction table access latency. Therefore, enlarging instruction streams will improve the behavior of a fetch engine based on streams. We provide a comprehensive analysis of dynamic instruction streams, showing that focusing on particular kinds of stream is not a good strategy due to Amdahl's law. Consequently, we propose the multiple stream predictor, a novel mechanism that deals with all kinds of streams by combining single streams into long virtual streams. We show that our multiple stream predictor is able to tolerate the prediction access latency without requiring the complexity caused by additional hardware mechanisms like prediction overriding.

1 Introduction

High performance superscalar processors require high fetch bandwidth to exploit all the available instruction-level parallelism. The development of accurate branch prediction mechanisms has provided important improvements in the fetch engine performance. However, it has also increased the fetch architecture complexity. Our approach to achieve high fetch bandwidth, while maintaining the complexity under control, is the stream fetch engine [19, 24].

This fetch engine design is based on the next stream predictor, an accurate branch prediction mechanism that uses instruction streams as the basic prediction unit. We call stream to a sequence of instructions from the target of a taken branch to the next taken branch, potentially containing multiple basic blocks. Figure 1 shows an example control flow graph from which we will find the possible streams. The figure shows a loop containing an if-then-else

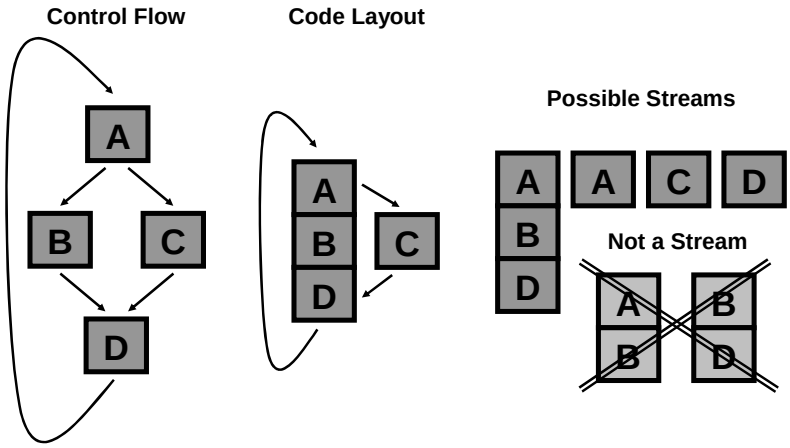


Figure 1. Example of instruction streams.

structure. Our profile data shows that $A \rightarrow B \rightarrow D$ is the most frequently followed path through the loop. Using this information, we lay out the code so that the path $A \rightarrow B$ goes through a not-taken branch, and falls-through from $B \rightarrow D$. Basic block C is mapped somewhere else, and can only be reached through a taken branch at the end of basic block A.

From the resulting code layout we may encounter four possible streams composed by basic blocks ABD, A, C, and D. The first stream corresponds to the sequential path starting at basic block A and going through the frequent path found by our profile. Basic block A is the target of a taken branch, and the next taken branch is found at the end of basic block D. Neither the sequence AB, nor the sequence BD can be considered streams because the first one does not end in a taken branch, and the second one does not start in the target of a taken branch. The infrequent case follows the taken branch at the end of A, goes through C, and jumps back into basic block D.

Although a fetch engine based on streams is not able to fetch instructions beyond a taken branch in a single cycle, streams are long enough to provide high fetch bandwidth. In addition, since streams are sequentially stored in the instruction cache, the stream fetch engine does not need a special-purpose storage, nor a complex dynamic building engine. However, taking into account current technology trends, accurate branch prediction and high fetch bandwidth is not enough. The continuous increase in processor clock frequency, as well as the larger wire delays caused by modern technologies, prevent branch prediction tables from being accessed in a single cycle [1, 13]. This limits fetch engine performance because each branch prediction depends on the previous one, that is, the target address of a branch prediction is the starting address of the following one.

A common solution for this problem is the prediction overriding technique [13, 26]. A small and fast predictor is used to obtain a first prediction in a single cycle. A slower but more accurate predictor provides a new prediction some cycles later, overriding the first prediction if they differ. This mechanism partially hides the branch predictor

access latency. However, it also causes an increase in the fetch architecture complexity, since prediction overriding requires a complex recovery mechanism to discard the wrong speculative work based on overridden predictions.

An alternative to the overriding mechanism is using long basic prediction units. A stream prediction contains enough instructions to feed the execution engine during multiple cycles [24]. Therefore, the longer a stream is, the more cycles the execution engine will be busy without requiring a new prediction. If streams are long enough, the execution engine of the processor can be kept busy during multiple cycles while a new prediction is being generated. Overlapping the execution of a prediction with the generation of the following prediction allows to partially hide the access delay of this second prediction, removing the need for an overriding mechanism, and thus reducing the fetch engine complexity.

Since instruction streams are limited by taken branches, a good way to obtain longer streams is removing taken branches through code optimizations. Code layout optimizations have a beneficial effect on the length of instruction streams [24]. These optimizations try to map together those basic blocks which are frequently executed as a sequence. Therefore, most conditional branches in optimized code are not taken, enlarging instruction streams. However, code layout optimizations are not enough for the stream fetch engine to completely overcome the need for an overriding mechanism [25].

Looking for novel ways of enlarging streams, we present a detailed analysis of dynamic instruction streams. We study the percentage of dynamic streams having a determinate length, showing that the majority of dynamic streams are shorter than the average. Thus, in order to increase the length of instruction streams, research should be focused on these short streams. Our results show that most of them finalize in conditional branches, function calls, and return instructions. Therefore, these kinds of branches are the best candidates to apply techniques for enlarging instruction streams.

In order to remove taken conditional branches, we present two new versions of the next stream predictor. For dealing with forward conditional branches, we present a version of the next stream predictor that includes taken conditional forward branches inside a stream when the number of instructions skipped by the forward branch is small. To deal with backward conditional branches, we present a loop stream predictor, i.e., a version of the stream predictor that is able to predict high-level loop structures, making it possible to combine all the iterations of simple loops into a single long virtual stream. Finally, regarding function calls and returns, we apply aggressive procedure inlining. This optimization replaces a procedure call by the procedure itself, removing the call and return instructions. Since both procedure calls and return instructions are taken branches, procedure inlining involves an increase in the length of instruction streams.

Unfortunately, these techniques do not provide good performance results. They tend to cover a small fraction of all dynamic streams or even to enlarge streams that are already long enough. It remains an important percentage of all dynamic streams that have just one or a few instructions. According to Amdahl's law, short streams that do

not benefit from the previous techniques severely limit the achievable performance improvement. This leads to a clear conclusion: the correct approach is not focusing on particular branch types, but trying to enlarge all dynamic streams. In order to achieve this, we present the multiple stream predictor, a novel predictor that concatenates those streams that are frequently executed as a sequence. This predictor does not depend on the type of the branch terminating the stream, making it possible to generate very long virtual streams.

The remainder of this paper is organized as follows. Section 2 describes previous related work. Section 3 presents our experimental methodology. Section 4 provides an analysis of dynamic instruction streams. Section 5 describes a technique for including short forward branches in instruction streams. Section 6 presents a novel loop stream predictor. Section 7 details the impact of aggressive procedure inlining on the length of instruction streams. Section 8 describes the multiple stream predictor. Finally, Section 9 presents our concluding remarks.

2 Related Work

The prediction table access latency is becoming an important limiting factor for current fetch architectures. The processor front-end must generate the fetch address in a single cycle because this address is needed for fetching instructions in the next cycle. However, the increase in processor clock frequency, as well as the slower wires in modern technologies, cause branch prediction tables to require multi-cycle accesses [1, 13].

The trace predictor [11] is a latency tolerant mechanism, since each trace prediction is potentially a multiple branch prediction. The processor front-end can use a single trace prediction to feed the processor back-end with instructions during multiple cycles, while the trace predictor is being accessed again to obtain a new prediction. Overlapping the prediction table access with the fetch of instructions from a previous prediction allows to hide the branch predictor access delay. Our next stream predictor has the same ability [25], since a stream prediction is also a multiple branch prediction able to provide enough instructions to hide the prediction table access latency.

Using a fetch target queue (FTQ) [20] is also helpful for taking advantage of this fact. The FTQ decouples the branch prediction mechanism and the instruction cache access. Each cycle, the branch predictor generates the fetch address for the next cycle, and a fetch request that is stored in the FTQ. Since the instruction cache is driven by the requests stored in the FTQ, the fetch engine is less likely to stay idle while the predictor is being accessed again.

Besides, there are two different approaches for tolerating the branch predictor access latency. On the one hand, code optimizations can be used to rearrange the code and increase the probability of overlapping branch prediction accesses with the execution of useful work. On the other hand, hardware mechanisms, like prediction overriding, can be used for reducing the negative impact of the predictor access latency. Both approaches are discussed in the following paragraphs.

2.1 Code Optimization

In order to hide the prediction table access latency by executing instructions from a prediction, it is important to increase the amount of instructions provided by each prediction, that is, enlarging the prediction unit. In the context of instruction streams, this can be achieved by minimizing the amount of taken branches. Code layout optimizations based on profile information can change the mapping of basic blocks, making it possible to map together those basic blocks that are frequently executed as a sequence. This technique leads to a code layout where most branches are not taken. Therefore, using code optimizers like Spike [6] can increase the length of instruction streams, and thus increase the stream predictor ability of tolerating access latency.

Another code optimization that can be used to reduce the amount of taken branches is procedure inlining. This optimization replaces a procedure call by the procedure itself, removing the call and return instructions. Procedure inlining is a frequently used code optimization. Allen and Johnson [2] describe a procedure inliner for C programs. However, they consider that only small procedures should be inlined to avoid an increase in the number of instruction cache misses. Hwu and Chang [10] present profile-driven algorithms for applying inlining and code reordering. The profile information is used to decide whether inlining a procedure will be beneficial for the program execution, allowing to inline bigger procedures. In addition, reordering the program code is an effective technique to alleviate the increase in the instruction cache misses caused by inlining big procedures.

Ayers et al. [4] describe an aggressive inliner, based on profile information, that is able to inline procedures at almost any call site without restriction. Their results show that aggressive inlining can provide important performance improvements in some benchmarks. Likewise, the ALTO [17] optimizer is able to aggressively inline procedures, using profile-based code reordering to reduce the negative effects of inlining on the instruction cache. Aydin and Kaeli [3] take this one step further implementing cache line coloring algorithms. They use ALTO to aggressively inline procedures, showing that cache line coloring is beneficial for reducing the negative impact of inlining on the instruction cache miss rate. In this paper, we show that instruction streams can be enlarged by removing taken branches using these techniques, that is, by removing function calls and returns, which makes it possible to tolerate larger predictor access latencies.

2.2 Hardware Mechanisms for Tolerating Access Latency

A promising idea to tolerate the prediction table access latency is pipelining the branch predictor [12, 27]. Using a pipelined predictor, a new prediction can be started each cycle. Nevertheless, this is not trivial, since the result of a branch prediction is needed to start the next prediction. Therefore, each branch prediction can only use the information available in the cycle it starts, which has a negative impact on prediction accuracy. In-flight information could be taken into account when a prediction is generated, as described in [27], but this also

involves an increase in the fetch engine complexity. It is possible to reduce this complexity in the fetch engine of a simultaneous multithreaded processor [30], as described by Falcon et al. [7], pipelining the branch predictor and interleaving prediction requests from different threads each cycle. Nevertheless, analyzing the accuracy and performance of pipelined branch predictors is out of the scope of this work.

A different approach is the overriding mechanism described by Jimenez et al. [13]. This mechanism provides two predictions, a first prediction coming from a fast branch predictor, and a second prediction coming from a slower, but more accurate predictor. When a branch instruction is predicted, the first prediction is used while the second one is still being calculated. Once the second prediction is obtained, it overrides the first one if they differ, since the second predictor is considered to be the most accurate. A similar mechanism is used in the Alpha EV6 [8] and EV8 [26] processors, where a multi-cycle latency branch predictor overrides a faster but less accurate cache line predictor [5].

The problem of prediction overriding is that it requires a significant increase in the fetch engine complexity. An overriding mechanism requires a fast branch predictor to obtain a prediction each cycle. This prediction should be stored for being compared with the main prediction. Some cycles later, when the main prediction is generated, the fetch engine should determine whether the first prediction is correct or not. If the first prediction is wrong, all the speculative work done based on it should be discarded. Therefore, the processor should track which instructions depend on each prediction done in order to allow the recovery process. This is the main source of complexity of the overriding technique.

Moreover, a wrong first prediction does not involve that all the instructions fetched based on it are wrong. Since both the first and the main predictions start in the same fetch address, they will partially coincide. Thus, the correct instructions based on the first prediction should not be squashed. This selective squash will increase the complexity of the recovery mechanism. To avoid this complexity, a full squash could be done when the first and the main predictions differ, that is, all instructions depending on the first prediction are squashed, even if they should be executed again according to the main prediction. However, a full squash will degrade the processor performance and does not remove all the complexity of the overriding mechanism. Therefore, the challenge is to develop a technique able to achieve the same performance than an overriding mechanism, but avoiding its additional complexity, which is one of the objectives of this work.

3 Experimental Methodology

The results in this paper have been obtained using trace driven simulation of a superscalar processor. Our simulator uses a static basic block dictionary to allow simulating the effect of wrong path execution. This model includes the simulation of wrong speculative predictor history updates, as well as the possible interference and

prefetching effects on the instruction cache. We feed our simulator with traces of 300 million instructions collected from the SPEC 2000 integer benchmarks[1] using the *reference* input set. To find the most representative execution segment we have analyzed the distribution of basic blocks as described in [28].

Since previous work [24] has shown that code layout optimizations are able to enlarge instruction streams, we present data for both a baseline and an optimized code layout. The baseline code layout was generated using the Compaq C V5.8-015 compiler on Compaq UNIX V4.0. The optimized code layout was generated with the Spike tool shipped with Compaq Tru64 Unix 5.1. Optimized code generation is based on profile information collected by the Pixie V5.2 tool using the *train* input set.

3.1 Inlining Methodology

The aggressive procedure inlining analysis presented in Section 7 uses the same simulation tool, but simulates different benchmark parts and uses a different code optimizer. Since the source code of our Spike version is not available for modifying the inlining heuristics, we generated the optimized code layout for the inlining analysis using the ALTO optimizer[2] [17] still using the profile information collected using the Pixie V5.2 tool and the *train* input set. ALTO allowed us to generate optimized binaries without inlining, as well as several sets of binaries with different levels of inlining.

In addition, since the aggressive procedure inlining analysis requires evaluating different sets of binaries, data presented in Section 7 have been obtained simulating all benchmarks until completion, which assures a fair comparison. Therefore, it is important to take into account that data presented for our inlining analysis is not comparable to data presented in other sections of this paper. In fact, the performance results presented in the inlining analysis are not measured in IPC but in execution cycles.

In order to explore a wide range of setups and binaries, we have chosen benchmark inputs from the *MinneSPEC* [16] input set. This input set has been especially designed to facilitate efficient simulations, avoiding excessively large simulation times. However, not all SPECint2000 benchmarks had a *MinneSPEC* input available when we selected our input set. In addition, we have discarded *MinneSPEC* inputs derived from the *train* input set. For those benchmarks that do not have an adequate *MinneSPEC* input available, we selected an input from the official *test* input set. Our benchmark input set is shown in Table 1.

[1] We excluded *181.mcf* because its performance is very limited by data cache misses, being insensitive to changes in the fetch architecture. We have thoroughly checked that including *181.mcf* does not change the conclusions of our work, but makes the plots harder to read.

[2] Our inlining analysis does not present data for the benchmark *252.eon* because our ALTO version is unable to optimize it.

benchmark	input	input set
164.gzip	input.random	minnespec
175.vpr	place	minnespec
176.gcc	cccp.s	test
186.crafty	crafty.in	test
197.parser	red.in	minnespec
253.perlbmk	makerand.pl	minnespec
254.gap	test.in	test
255.vortex	persons.1k	test
256.bzip2	input.source	minnespec
300.twolf	test	test

Table 1. Benchmark suite used and the corresponding input set.

fetch, rename, and commit widths	8 instructions
integer and floating point issue widths	8 instructions
load/store issue width	4 instructions
fetch target queue	4 entries
instruction fetch queue	32 entries
integer, floating point, and load/store issue queues	64 entries
reorder buffer	256 entries
integer and floating point registers	160
L1 instruction cache	64 or 32 Kbytes, 2-way associative, 128 byte block
L1 data cache	64 Kbytes, 2-way associative, 64 byte block
L2 unified cache	1 Mbyte, 4-way associative, 128 byte block, 10 cycle latency
main memory latency	100 cycles
maximum trace size	32 instructions (10 branches)
filter and main trace caches	128 traces, 4-way associative

Table 2. Configuration of the simulated processor

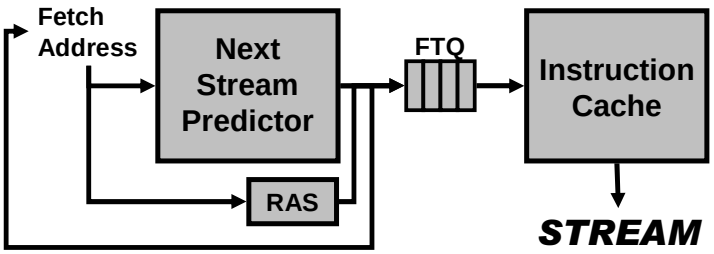
3.2 Simulator Setup

Our simulation setup corresponds to an aggressive 8-wide superscalar processor. The main values of this setup are shown in Table 2. We compare our stream fetch architecture with three other state-of-the-art fetch architectures: a fetch architecture using an interleaved BTB and a 2bcgskew predictor [26], the fetch target buffer (FTB) architecture [20] using a perceptron predictor [14], and the trace cache fetch architecture using a trace predictor [11]. All these architectures use a 4-entry fetch target queue (FTQ) [20] to decouple the branch prediction stage from the fetch stage. We have found that larger FTQs do not provide additional performance improvements.

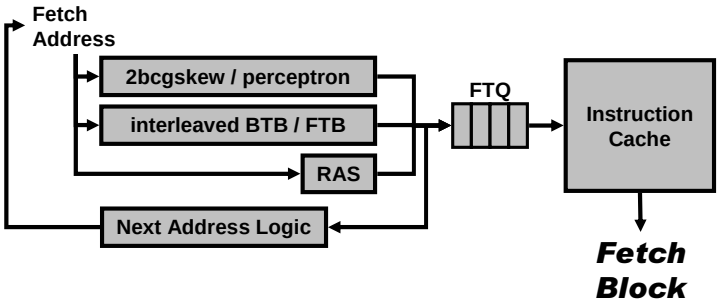
Our instruction cache setup uses wide cache lines, that is, 4 times the processor fetch width [19], and a 64KB total hardware budget. The trace fetch architecture is actually evaluated using a 32KB instruction cache, while the remainder 32KB are devoted to the trace cache. This hardware budget is equally divided into a filter trace cache [21] and a main trace cache. In addition, we use selective trace storage [18] to avoid trace redundancy between the trace cache and the instruction cache.

3.3 Fetch Models

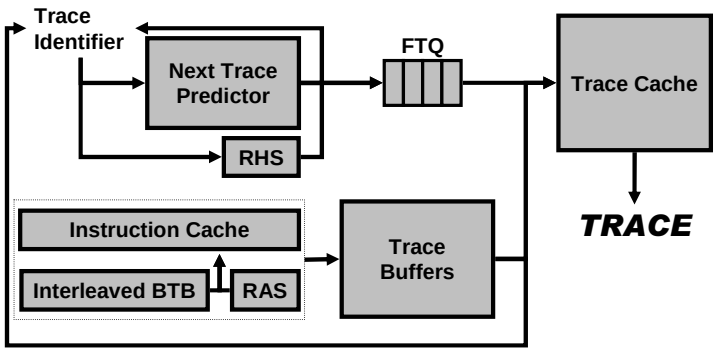
The stream fetch engine [19, 24] model is shown in Figure 2.a. The stream predictor access is decoupled from the instruction cache access using an FTQ. The stream predictor generates requests, composed by a full stream



(a) The stream fetch engine



(b) Fetch engine using a BTB/FTB and a decoupled conditional branch predictor



(c) Trace cache fetch architecture using a next trace predictor

Figure 2. Fetch models evaluated.

of instructions, which are stored in the FTQ. These requests are used to drive the instruction cache, obtain a line from it, and select which instructions from the line should be executed. In the same way, the remainder three fetch models use an FTQ to decouple the branch prediction stage from the fetch stage.

Our interleaved BTB fetch model (iBTB) is inspired by the EV8 fetch engine design described in [26]. Our iBTB model decouples the branch prediction mechanism from the instruction cache with an FTQ. An interleaved BTB is used to allow the prediction of multiple branches until a taken branch is predicted, or until an aligned 8-instruction block is completed. The branch prediction history is updated using a single bit for prediction block, which combines the outcome of the last branch in the block with path information [26]. Our FTB model is similar to the one described in [20] but using a perceptron branch predictor [14] to predict the direction of conditional branches. Figure 2.b shows a diagram representing these two fetch architectures.

Our trace cache fetch model is similar to the one described in [22] but using an FTQ [20] to decouple the trace predictor from the trace cache, as shown in Figure 2.c. Trace predictions are stored in the FTQ, which feeds the trace cache with trace identifiers. An interleaved BTB is used to build traces in the case of a trace cache miss. This BTB uses 2-bit saturating counters to predict the direction of conditional branches when a trace prediction is not available. In addition, an aggressive 2-way interleaved instruction cache is used to allow traces to be built as fast as possible. This mechanism is able to obtain up to a full cache line in a cycle, independent of PC alignment.

The four fetch architectures evaluated in this paper use specialized structures to predict return instructions. The iBTB, the FTB, and the stream fetch architecture use a return address stack (RAS) [15] to predict the target address of return instructions. There are actually two RAS, one updated speculatively in prediction stage, and another one updated non-speculatively in commit stage, which is used to restore the correct state in case of a branch misprediction. The iBTB and FTB fetch architectures also use a cascaded structure [23] to improve the prediction accuracy of the rest of indirect branches. Both the stream predictor and the trace predictor are accessed using correlation, and thus they are already able to correctly predict indirect jumps and function calls.

The trace fetch architecture uses a return history stack (RHS) [11] instead of a RAS. This mechanism is more efficient than a RAS in the context of trace prediction because the trace predictor is indexed using a history of previous trace identifiers instead of trace starting addresses. There are also two RHS, one updated speculatively in prediction stage, and another one updated non-speculatively in commit stage. However, the RHS in the trace fetch architecture is less accurate predicting return instructions than the RAS in the rest of evaluated architectures. Trying to alleviate this problem, we also use a RAS to predict the target address of return instructions during the trace building process.

3.4 Branch Prediction Setup

We have evaluated the four simulated fetch engines varying the size of the branch predictor from small and fast tables to big and slow tables. We use realistic prediction table access latencies calculated using the CACTI 3.0 tool [29]. We modified CACTI to model tagless branch predictors, and to work with setups expressed in bits instead of bytes. Data we have obtained corresponds to 0.10μm technology. For translating the access time from nanoseconds to cycles, we assumed an aggressive 8 fan-out-of-four delays clock period, that is, a 3.47 GHz clock frequency as reported in [1]. It has been claimed in [9] that 8 fan-out-of-four delays is the optimal clock period for integer benchmarks in a high performance processor implemented in 0.10μm technology.

We have found that the best performance is achieved using three-cycle latency tables [25]. Although bigger predictors are slightly more accurate, their increased access delay harms processor performance. On the other hand, predictors with a lower latency are too small and provide a poor performance. Therefore, we have chosen to simulate all branch predictors using the bigger tables that can be accessed in three cycles. Table 3 shows the

iBTB fetch architecture (approx. 95KB)		
2bcgskew predictor	interleaved BTB	1-cycle predictor
four 64K entry tables 16 bit history (bimodal 0 bits)	1024 entry, 4-way, first level 4096 entry, 4-way, second level DOLC 14-2-4-10	64 entry gshare 6-bit history 32 entry, 1-way, BTB
FTB fetch architecture (approx. 50KB)		
perceptron predictor	FTB	1-cycle predictor
256 perceptrons 4096x14 bit local and 40 bit global history	1024 entry, 4-way, first level 4096 entry, 4-way, second level DOLC 14-2-4-10	64 entry gshare 6-bit history 32 entry, 1-way, BTB
Stream fetch architecture (approx. 32KB)		
next stream predictor		1-cycle predictor
1024 entry, 4-way, first level 4096 entry, 4-way, second level DOLC 16-2-4-10		32 entry, 1-way, spred DOLC 0-0-0-5
Trace fetch architecture (approx. 80KB)		
next trace predictor	interleaved BTB	1-cycle predictor
2048 entry, 4-way, first level 4096 entry, 4-way, second level DOLC 10-4-7-9	1024 entry, 4-way, first level 4096 entry, 4-way, second level DOLC 14-2-4-10	32 entry, 1-way, tpred DOLC 0-0-0-5 perfect BTB override

Table 3. Configuration of the simulated branch predictors.

configuration of the simulated predictors. We have explored a wide range of history lengths, as well as DOLC index [11] configurations, and selected the best one found for each setup. Table 3 also shows the approximated hardware budget for each predictor. Since we simulate the larger three cycle latency tables[3], the total hardware budget devoted to each predictor is different. The stream fetch engine requires less hardware resources because it uses a single prediction mechanism, while the other evaluated fetch architectures use some separate structures.

Our fetch models also use an overriding mechanism [13, 26] to complete a branch prediction each cycle. A small branch predictor, supposed to be implemented using very fast hardware, generates the next fetch address in a single cycle. Although being fast, this predictor has a low accuracy, so the main predictor is used to provide an accurate back-up prediction. This prediction is obtained three cycles later and compared with the prediction provided by the single-cycle predictor. If both predictions differ, the new prediction overrides the previous one, discarding the speculative work done based on it. The configuration of the single-cycle predictors used is shown in Table 3.

4 Analysis of Dynamic Instruction Streams

Fetching a single basic block per cycle is not enough to keep busy the execution engine of wide-issue superscalar processors during multiple cycles. In this context, the main advantage of instruction streams is their long size. A stream can contain multiple basic blocks, whenever only the last one ends in a taken branch. This makes it possible for the stream fetch engine to provide high fetch bandwidth while maintaining the implementation cost under control.

[3]The first level of the trace and stream predictors, as well as the first level of the cascaded iBTB and FTB, is actually smaller than the second one because larger first level tables do not provide a significant improvement in prediction accuracy.

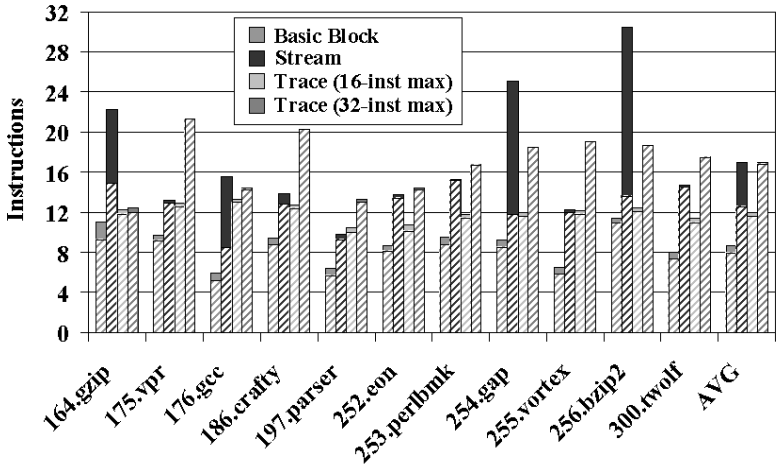


Figure 3. Average length of basic blocks, instruction streams, and instruction traces for both baseline codes (shadowed bar) and optimized codes (full bar).

4.1 The Length of Instruction Streams

Figure 3 shows the average length of dynamic basic blocks and dynamic instruction streams. The shadowed part of each bar shows data using our baseline code layout. On average, instruction streams are 55% longer than basic blocks. This fact allows the stream fetch engine to outperform other fetch architectures based on basic blocks, as shown in [24], while requiring similar or even lower complexity.

Figure 3 also shows the average length of dynamic instruction traces. The advantage of the trace cache fetch architecture is that it can fetch instructions beyond a taken branch in a single cycle. However, since traces are stored in a special purpose cache, their size is physically limited. Using a maximum trace size of 16 instructions [22] and our baseline code layout, streams are, on average, 8% longer than traces. Increasing the maximum trace size to 32 instructions involves an increase in the average trace length. Although traces are also limited by other factors, like indirect branches, the average trace length becomes 32% longer than the average stream length. The drawback of increasing the maximum trace size is that it reduces the total number of traces that can be stored in the trace cache, limiting performance. In general, as shown in [24], streams are long enough to provide a performance similar to a trace cache at a lower complexity.

The full bars at Figure 3 show the average length of dynamic basic blocks, streams, and traces using optimized codes. Code layout optimizations try to map together those basic blocks that are frequently executed as a sequence. Therefore, most dynamic conditional branches in optimized codes are not taken, enlarging instruction streams. However, the length of basic blocks and traces does not benefit from this effect. This happens because basic blocks contain a single branch instruction, despite the branch is taken or not, while traces are not limited by taken conditional branches.

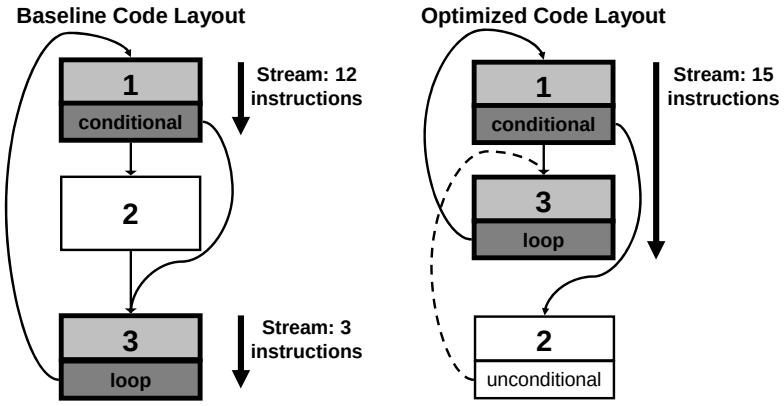


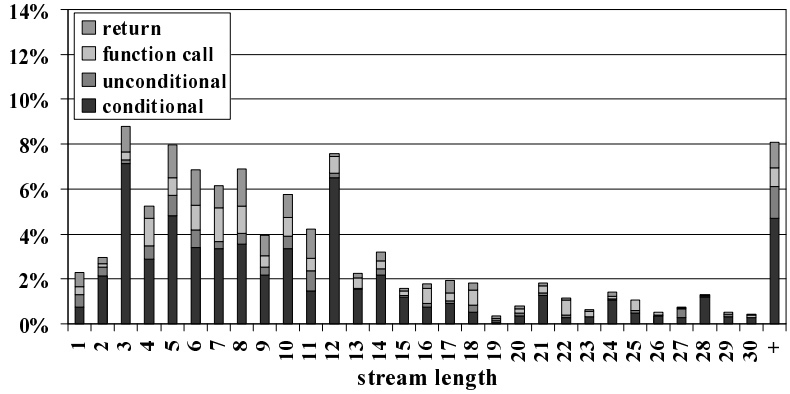
Figure 4. Example of code layout optimization taken from the *176.gcc* benchmark.

Figure 4 shows an example of code optimization taken from the *176.gcc* benchmark. A 12-instruction basic block ends in a conditional branch. When this branch is taken, it goes to a 3-instruction basic block that ends in a loop branch. This loop goes to the first basic block when it iterates. Using the baseline code layout, this structure contains two instruction streams, each of one representing 27% of the total number of dynamic streams in the program. The portion of code between these two streams is rarely executed. Using the optimized code layout, the rarely executed code has been laid out in other place, mapping together the two frequently executed basic blocks. Thus, the optimized code has a single 15-instruction stream which is responsible for 54% of the total number of streams.

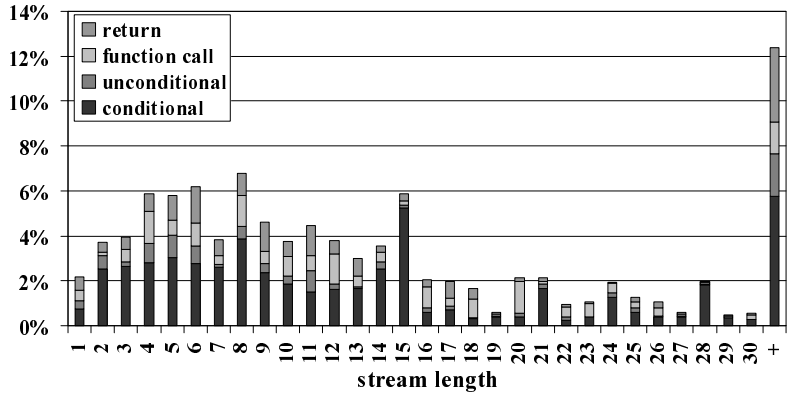
The increase in the average stream length achieved by using optimized codes is beneficial for the next stream predictor accuracy. Having longer streams causes most part of the program execution to be held in a lower number of streams. This fact reduces aliasing in the prediction table, increasing prediction accuracy. Having longer streams also involves an increase in fetch bandwidth, which improves the stream fetch engine performance, allowing it to feed wider execution cores. Using optimized codes, instruction streams have an average length very close to 32-instruction maximum traces. In addition streams are, on average, 40% longer than traces and 95% longer than basic blocks when using optimized codes. This allows the stream fetch engine to provide a performance even closer to the trace cache fetch architecture.

4.2 Distribution of Stream Lengths

Longer streams make it possible for the stream fetch engine to achieve better performance. However, having a long average length does not involve that most streams are long. Some streams could be long, providing high fetch bandwidth, while other streams could be short, degrading the potential performance. Therefore, in the search for new ways of improving the stream fetch engine performance, the distribution of dynamic stream lengths should be analyzed.



(a) baseline code



(b) optimized code

Figure 5. Histograms of dynamic streams classified according to their length and the terminating branch type. The results presented in these histograms are the average of the eleven benchmarks used.

Figure 5 shows an histogram of dynamic streams classified according to their length. It shows the percentage of dynamic streams that have a length ranging from 1 to 30 instructions. The last bar shows the percentage of streams that are larger than 30 instructions. Data is shown for both the baseline and the optimized code layout. In addition, streams are divided according to the terminating branch type: conditional branches, unconditional branches, function calls, and returns.

Using the baseline code layout, most streams are shorter than the average length: 70% of the dynamic streams have 12 or less instructions. Using the optimized code layout, the average length is longer. However, most streams are still shorter than the average length: 70% of the dynamic streams have 15 or less instructions. Therefore, in order to increase the average stream length, research should be focused in those streams that are shorter than the average length. For example, if we consider an 8-wide execution core, research effort should be devoted to enlarge streams shorter than 8 instructions. Using optimized codes, the percentage of those streams is reduced from 40% to 30%. Nevertheless, there is still room for improvement.

Most dynamic streams finish in taken conditional branches. They are 60% when using the baseline code and 52% when using the optimized code. The percentage is lower in the optimized codes due to the higher number of not taken conditional branches, which never finish instruction streams. There also is a big percentage of streams terminating in function calls and returns. They are 30% of all dynamic streams in the baseline code. The percentage is larger in the optimized code: 36%. This happens because code layout optimizations are mainly focused on conditional branches. Since the number of taken conditional branches is lower, there is a higher percentage of streams terminating in other types of branches, although the total number is similar. This means that, in order to enlarge instruction streams, the most promising field for research are conditional branches, function calls, and return instructions.

5 Including Short Forward Branches in Instruction Streams

Most dynamic streams are finished by taken conditional branches. Enlarging streams beyond these branches would allow an improvement in the fetch bandwidth, as well as an increase in branch prediction accuracy and a reduction in the branch predictor energy consumption. In this section, we describe a technique that includes forward conditional branches in instruction streams.

5.1 Short Forward Branches

Our proposal is aimed to reduce the impact of a possible misprediction without requiring an increase in the fetch engine complexity. The idea is selecting those conditional branch instructions whose target is a few instructions ahead in the code layout, that is, short forward branches (SFB). When a taken SFB is found, the instruction stream is not finished by it. In spite of this, the SFB is included in the stream, which will be finished by the next taken branch.

When such a stream is predicted, both the branch terminating the stream and the included SFB are implicitly predicted taken. The front-end fetches all instructions indicated by the stream prediction, but the instructions branched by the SFB are masked in order to avoid their execution. In case of an SFB misprediction, the speculative processor state should be squashed. However, the instructions from the correct path, i.e. the instructions branched by the SFB, have already been fetched. In this way, we are reducing the impact of SFB mispredictions, also avoiding the need for fetching again the instructions after the SFB target.

Figure 6 shows an example: a loop containing an if-then structure, as well as its code layout. According to our traditional stream definition, we may find three possible streams composed by basic blocks ABC, A, and C. Let us now suppose that the conditional branch terminating basic block A is an SFB. Let us also suppose that the processor knows (e.g. using profiling) that it has a high probability of being mispredicted. In this case, a new

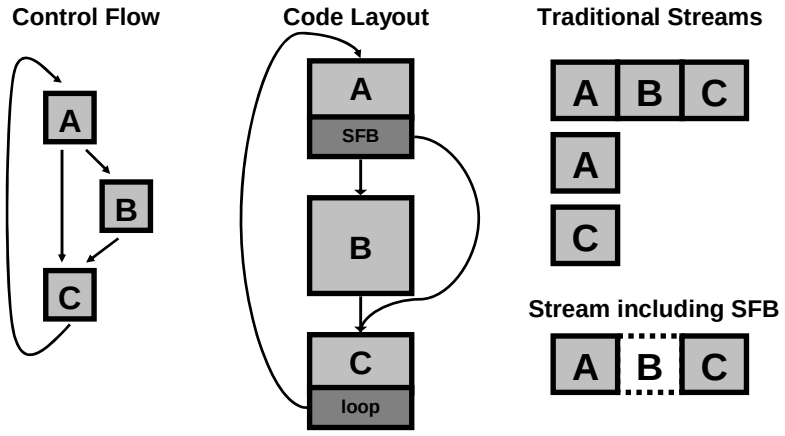


Figure 6. Example of instruction streams containing an SFB.

stream appears, composed by basic blocks ABC, which contains additional information indicating that basic block B should not be executed. When this stream is predicted, the three basic blocks will be fetched, although only basic blocks A and C will be executed. This means that the SFB terminating basic block A is predicted taken. If it is mispredicted, the speculative work based on basic block C is discarded, but the processor does not need to fetch it again, since the correct path after the SFB, that is, basic block B, has already been fetched.

The main drawback of this mechanism is that, if an SFB is not mispredicted, useless instructions have been fetched, wasting fetch bandwidth. Therefore, it is important to choose those branches which branch over a small number of instructions, and are very likely to be mispredicted. In addition, the instructions branched by an SFB should not contain another branch instruction, since such a branch will be implicitly predicted not taken. If it is taken, it will be mispredicted, not only reducing prediction accuracy, but also preventing the processor from taking advantage of the additional fetched instructions.

Figure 7 shows an example of a conditional branch that should not be considered an SFB. The structure shown in this figure is a loop containing an if-then construct. The if-then structure contains a break statement that causes an immediate exit from the loop when executed. The conditional branch terminating basic block A is not an SFB because the portion of code branched by it contains another branch, i.e. the break statement. If the branch terminating basic block A is considered an SFB and mispredicted, basic blocks B and C will be executed sequentially, which involves mispredicting the unconditional branch associated with the break statement.

5.2 Viability Evaluation

The potential of our SFB technique depends on the amount of branches that can take advantage of it. These branches are forward conditional branches with no other branch instructions inside the static code from the branch itself to its target, that is, the portion of code skipped when the branch is taken. This kind of branches corresponds

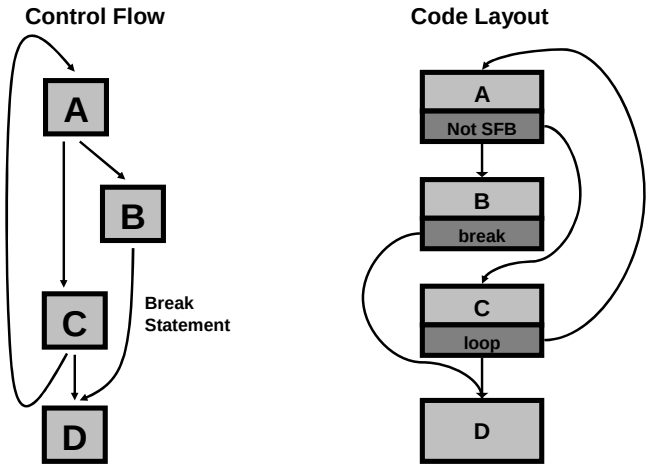


Figure 7. Example of a conditional branch that is not an SFB.

to if-then high level structures. Figure 8 shows an example of such a structure. The conditional branch terminating basic block A has basic block C as its taken target. If basic block B does not contain any branch instruction, the branch terminating basic block A can take advantage of our SFB mechanism. Since this structure is composed of three basic blocks, we call it a *hammock3* structure.

An if-then-else high level structure, like the one shown in Figure 9, can also take advantage of our SFB mechanism. The main difference between this structure and a *hammock3* structure is that the instruction terminating basic block B is an unconditional direct branch, which jumps over basic block C. If the branch terminating basic block A is taken, then basic block C will be executed; otherwise, basic block B will be executed. This kind of structure can also take advantage of our SFB mechanism, ignoring the unconditional branch terminating basic block B, whenever basic block B does not contain more branches, and basic block C does not contain any branch instruction. Since this structure is composed of four basic blocks, we call it a *hammock4* structure.

We have analyzed the distribution of dynamic branch instructions, looking for *hammock3* and *hammock4* structures. Table 4 shows the classification we have developed. It also shows the percentage of dynamic branches and dynamic branch mispredictions corresponding to each kind of branches. The results shown are the average of our eleven benchmarks using the baseline code layout. We have omitted results using optimized codes because code layout optimizations tend to remove *hammock3* and *hammock4* structures. The optimization process should be done taking into account the needs of our SFB mechanism, but this is out of the scope of this analysis.

Branch instructions are divided into forward conditional branches and other branches. Forward conditional branches are classified according to the number of branch instructions inside the portion of code skipped when the branch is taken: zero, one, two, three, and more than three. Those branches that do not contain any branch inside the skipped code are *hammock3* structures.

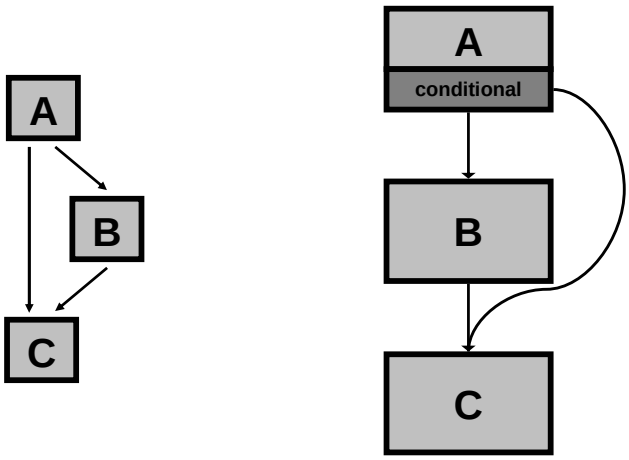


Figure 8. Code layout of a *hammock3* structure, that is, an if-then structure (without an else clause).

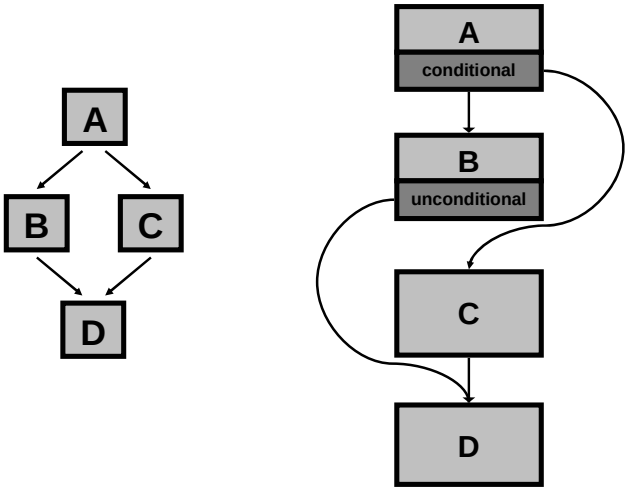


Figure 9. Code layout of a *hammock4* structure, that is, an if-then-else structure.

If a branch instruction contains a single branch inside the skipped code, it could be a *hammock4* structure whenever the branch contained inside is an unconditional direct branch. In this case, the branch is a *hammock4* if both the unconditional branch is the last instruction of the skipped code, and the portion of code jumped by the unconditional branch does not contain any additional branch instruction. Otherwise, the branch is classified as containing an unconditional branch. Branches containing a single branch inside the skipped code can also be classified as containing a conditional branch, a function call, or a return instruction.

Our results show that 2.5% of the dynamic branches correspond to *hammock3* structures, while 2.7% of the dynamic branches correspond to *hammock4* structures. This means that only 5.2% of the dynamic branches can benefit from our SFB mechanism. Regarding dynamic mispredictions, 6.3% of the mispredictions are caused by *hammock3* branches, while 6.4% of the mispredictions are caused by *hammock4* branches. Therefore, the current version of our SFB mechanism can only be used to reduce the impact of 12.7% of the dynamic mispredictions, and thus it has little potential to improve processor performance.

			branches	mispredictions
forward conditional branches	**no branches inside (hammock3)**		**2.5%**	**6.3%**
	one branch inside	**hammock4**	**2.7%**	**6.4%**
		unconditional	3.5%	6.7%
		conditional	5.6%	8.8%
		function call	3.3%	1.8%
		return	1.7%	3.4%
	two branches inside		6.7%	15.9%
	three branches inside		4.9%	4.9%
	more than three branches inside		12.4%	18.8%
other branches			56.7%	27.0%

Table 4. Distribution of dynamic branches and dynamic mispredictions according to our classification.

6 Loop Stream Prediction

The technique described in the previous section is only able to enlarge streams finished by forward conditional branches. Now, we present a technique aimed to enlarge streams finished by backward conditional branches. In particular, we try to enlarge loop streams, that is, instruction streams finalizing in a loop branch whose target is the starting address of the stream itself.

6.1 The Loop Stream Predictor

Figure 10 shows an example of loop stream. The structure shown is a loop containing an if-then structure, as well as its code layout. If the conditional branch finalizing basic block A is not taken, then the whole loop body is covered by a single instruction stream composed by basic blocks AC. In addition, when the loop iterates, the stream target is basic block A, that is, the starting address of the stream itself. We call this stream a loop stream.

The objective of our loop stream predictor is to combine the different iterations of a loop stream into a single long virtual stream. Given basic block A as fetch address, the original next stream predictor [19, 24] is able to predict the stream AC and basic block A as its target address. If this predictor is enhanced with the ability of predicting the number of iterations of the loop stream, the prediction achieved would be longer. Let us suppose that the stream predictor indicates that the loop composed by basic blocks AC will be executed four times. In this case, the stream predictor does not need to predict AC four times, since it can predict $ACACACAC$. In addition, the loop stream can be enlarged taking into account the loop exit. When the loop branch is not taken, the processor will execute the stream composed by basic block D, whose target address is elsewhere out of the loop (not shown in the figure). Therefore, given basic block A as fetch address, the loop stream predictor is able to predict stream $ACACACACD$, which is composed by 9 basic blocks.

In order to achieve this, we should add new fields to the stream prediction tables, as shown in Figure 11. The original stream predictor contains a length field and a target address field. The length field indicates the length of the loop stream, but the target address field is not required to indicate the target of the loop, since it is implicitly predicted as the starting address of the stream. The target address field will now be used to predict the target

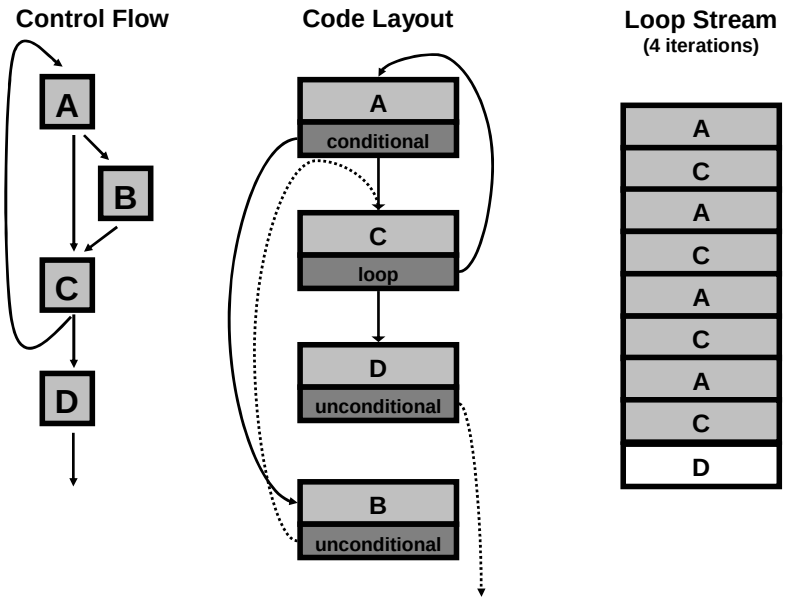


Figure 10. Example of loop stream.

address of the stream that appears after the loop exit, which involves that an additional length field is required to predict the length of this exit stream. Finally, a counter is needed to keep the number of iterations of the loop stream. This counter also indicates when a loop stream should be predicted or not: a stream prediction is a loop stream prediction whenever the number of iterations is not zero; otherwise it is a normal stream prediction.

Therefore, we only need to add an additional length field and an iteration counter. We have measured using CACTI [29] that adding these fields has no negative impact on the prediction table access latency. We should also add a small set of registers to keep loop stream predictions. When a loop stream is predicted, the FTQ is feed by the loop stream prediction registers. The length field is used to form the loop stream, which has the starting address of the stream (the tag field) as target. The iteration counter states how many times this loop stream should be introduced in the FTQ. Finally, the exit length and the target address are combined to generate the stream that should be executed after the loop stream, that is, when all iterations have been fetched.

While the loop prediction is active, there is no need to generate new stream predictions, reducing the overall branch prediction energy consumption and avoiding problems with the prediction table access latency. Once all the information in the loop stream prediction registers has been used to feed the FTQ, the stream predictor starts generating new predictions again until the next loop stream prediction is found.

6.2 Loop Stream Prediction Evaluation

Figure 12 shows the percentage of the total number of executed streams that are loop streams. As can be expected, there is a larger percentage when using optimized codes, since it is less likely that a taken branch

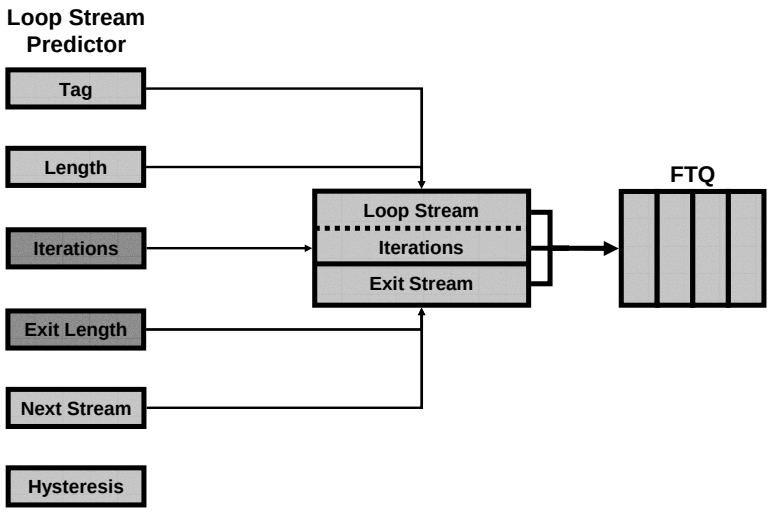


Figure 11. The loop stream predictor.

appears inside a loop body, breaking the loop stream. However, the average percentage of loop streams is not very high: 7% using the baseline code and 14% using the optimized code. Only the benchmark *176.gcc* contains a high percentage of loop streams when optimized. This is mainly due to the high-level structure described in Section 4 (Figure 4), which constitutes a loop stream that covers 54% of the total number of streams.

However, the main problem of our loop stream predictor is not the low portion of code covered by loop stream. Figure 13 shows the average length of loop streams. On average, this kind of streams have almost 30 instructions for both code layouts. For instance, combining 10 iterations of a 30-instruction loop stream would generate a prediction containing 300 instructions. Such a long prediction makes it possible a reduction in the branch predictor energy consumption, but will not improve the ability of tolerating the prediction table access latency, since a 30-instruction stream is long enough to hide the predictor access latency during several cycles, even for wide processors.

Consequently, loop streams do not need additional mechanisms for tolerating the prediction table access latency, and thus little performance improvement should be expected from our loop stream prediction technique. We have evaluated a wide range of loop predictor setups, and we have found that the best one achieves a prediction accuracy very similar to the original stream predictor. However, we have measured that, when loop stream prediction is enabled, a processor using a realistic access latency stream predictor achieves less than 1% performance speedup in most cases. Moreover, we have found using CACTI [29] that the reduction in the stream predictor activity does not compensate the increase in the prediction table size caused by the new fields required. The loop stream predictor consumes 0.5% more energy than the original stream predictor when using optimized codes, and 8% more energy when using baseline codes. These results suggest that, in order to enlarge instructions streams, different research lines should be examined.

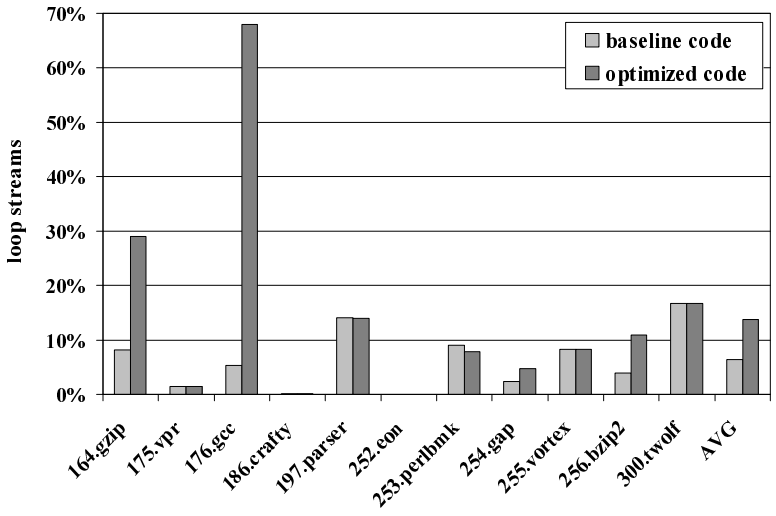


Figure 12. Percentage of dynamic instruction streams that are loop streams.

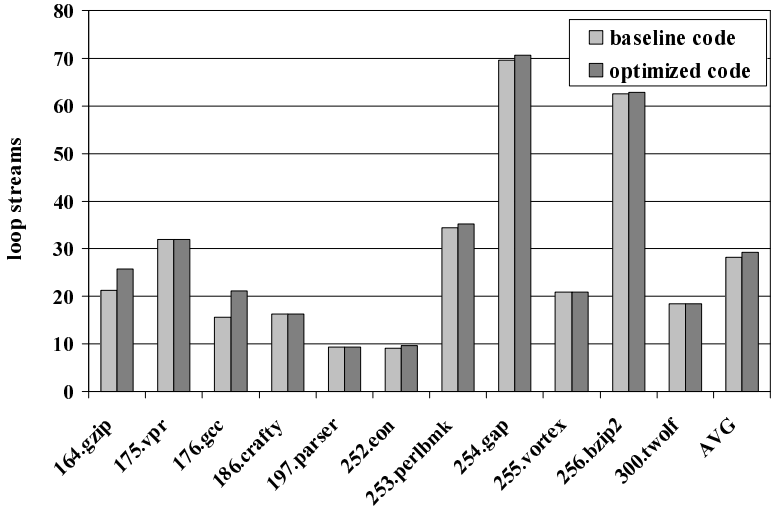


Figure 13. Average loop stream length.

7 Aggressive Procedure Inlining

The ALTO [17] optimizer is able to perform aggressive procedure inlining. This inlining optimization is mainly controlled by the maximum resultant code size (MRCS), that is, the maximum number of instructions that an inlined portion of code should have. A procedure is never inlined if the resultant code size is higher than MRCS. If the inlined procedure is called from a loop, then the number of instructions belonging to the loop plus the number of instructions belonging to the inlined procedure cannot be higher than MRCS. Otherwise, the number of instructions belonging to the caller procedure plus the number of instructions belonging to the inlined procedure cannot be higher than MRCS.

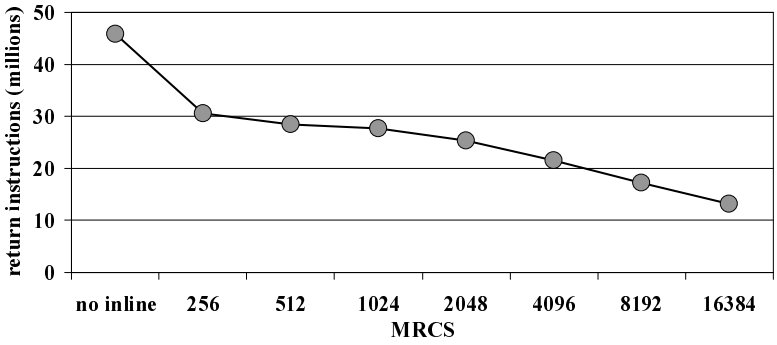


Figure 14. Average number of return instructions varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

The higher the MRCS value is, the more aggressive is the procedure inlining performed. As a measure of procedure inlining effectivity, Figure 14 shows the average number of return instructions. This number is equivalent to the average number of executed procedures. We evaluate the MRCS parameter varying from 256 to 16384 instructions, and compare it against our baseline code, that is, code optimized using ALTO without inlining. Higher values of MRCS involve a more aggressive inlining, and thus a higher reduction in the total number of return instructions. The more aggressive inlining provides a reduction over 70% of return instructions against the code optimized without inlining.

7.1 The Impact of Inlining on the Length of Instruction Streams

The aggressivity of procedure inlining has a direct impact on the length of instruction streams, as shown in Figure 15. By definition, instruction streams are limited by taken branches. Procedure inlining removes a big amount of taken branches: function calls and return instructions. The reduction in the number of taken branches involves an enlargement of instruction streams. This enlargement is limited by the removal of instructions associated with the procedure call overhead. Nevertheless, the overall effect is that the more aggressive the inlining is, the longer instruction streams are.

However, this enlargement of instruction streams is not for free. Aggressive inlining duplicates big amounts of the program code, increasing the number of instruction cache misses. Figure 16 shows the average number of instruction cache misses varying the MRCS value from 256 to 16384 instructions, and the total instruction cache hardware budget from 8KB to 64KB. Although, in general, more aggressive inlining involves a higher number of instruction cache misses, this is not always a direct relationship. For example, the 8KB instruction cache has a lower number of misses using a 1024-instruction MRCS value than using a 512-instruction MRCS value.

This happens because increasing the code size is not the only effect caused by procedure inlining. As mentioned before, inlining eliminates the instructions associated with the calling overhead. Aggressive inlining involves the

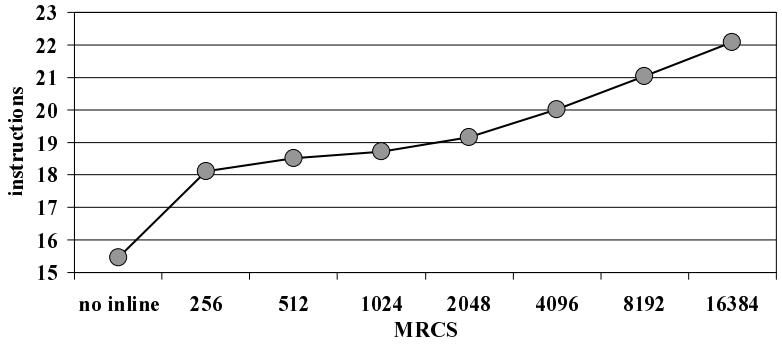


Figure 15. Average stream length varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

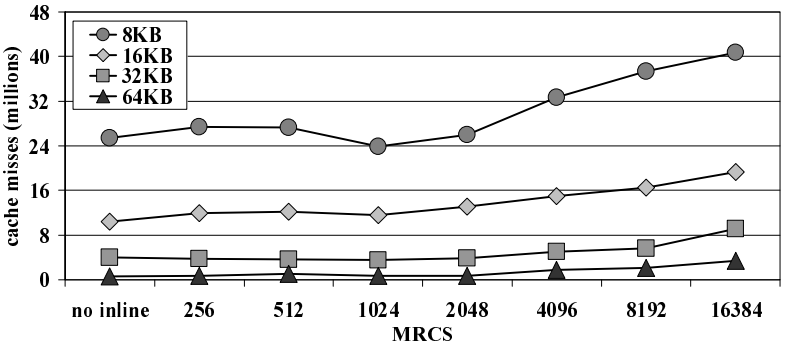


Figure 16. Average instruction cache misses varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

removal of a higher number of these instructions, limiting the increase in the number of instruction cache misses. Moreover, inlining removes procedure boundaries, increasing the visibility of the code to other optimizations, like dead code elimination or code scheduling, potentially reducing the number of instruction cache misses. Nevertheless, this only happens for intermediate values of the MRCS parameter, that is, when the impact on the cache miss rate is not too high. The higher values of MRCS always cause a higher number of instruction cache misses.

To summarize, aggressive inlining involves enlarging instruction streams, increasing the ability of the next stream predictor of tolerating the prediction table access latency, and thus improving the processor performance. On the other hand, aggressive inlining causes an increase in the number of instruction cache misses, which degrades the processor performance. In order to find the optimal setup, we explore this tradeoff in the next section.

7.2 Inlining Evaluation

Both the length of instruction streams and the number of instruction cache misses have an important impact on the overall processor performance. In this section, we evaluate the processor performance looking for a balance between the average stream length and the instruction cache miss rate.

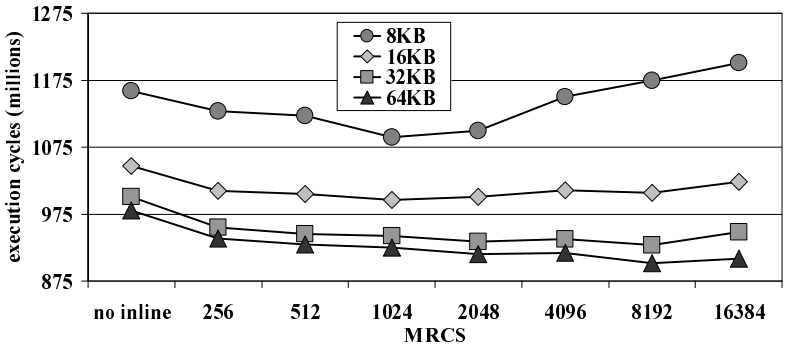


Figure 17. Average performance of an 8-wide processor varying the maximum resultant code size (MRCS) from 256 to 16384 instructions.

7.2.1 Overall Processor Performance

Figure 17 shows the processor performance using a realistic prediction table access latency (3 cycles) without prediction overriding. We vary the MRCS value from 256 to 16384 instructions, and the total instruction cache hardware budget from 8KB to 64KB. The bigger the cache is, the more aggressive is the inlining that can be performed. Thus, the optimal value of MRCS is higher for the bigger cache sizes. Both the 8KB and 16KB instruction caches achieve their optimal performance using a 1024-instruction MRCS value, while the 32KB and the 64KB instruction caches achieve their optimal performance using a more aggressive 8192-instruction MRCS value.

The best performance is achieved by the 64KB instruction cache due to its lower miss rate. Using this cache setup, the code inlined using the optimal MRCS value achieves 8% performance improvement over the non-inlined code. However, this improvement is not necessarily caused by the ability of tolerating the predictor access latency. It can also be caused by the higher fetch bandwidth provided by longer streams and the additional code optimizations enabled by our aggressive inlining. In order to provide more insight about this, we have measured the performance achieved by a processor with an ideal 1-cycle latency predictor, where overriding has no impact on performance.

Using the ideal latency predictor, the code inlined using the optimal MRCS value achieves 6% performance improvement over the non-inlined code. Since the predictor access latency is not a problem for the ideal latency predictor, this improvement cannot be attributed to the ability of tolerating the access delay, but to the higher fetch bandwidth and the additional code optimizations enabled by inlining. However, the performance improvement achieved by the realistic latency predictor is higher. Therefore, this additional 2% performance improvement is caused not by the improved fetch bandwidth and the additional code optimizations, but by the ability of longer streams of hiding the prediction table access latency.

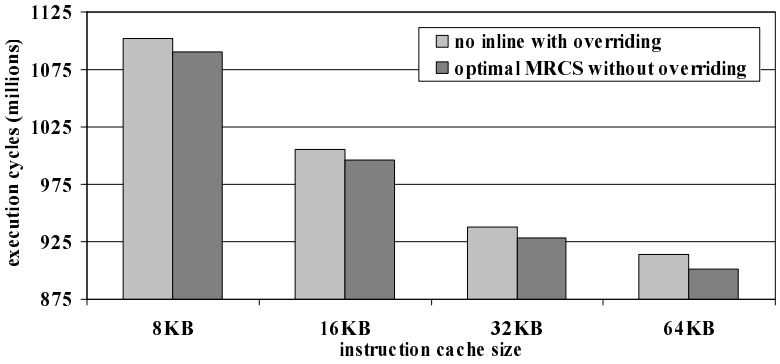


Figure 18. Average performance of an 8-wide processor with prediction overriding not using inlining, and without prediction overriding but using inlining and the optimal MRCS value.

Figure 18 shows the performance achieved using the optimal MRCS value for each instruction cache size. As mentioned before, these results correspond to a realistic prediction table delay without overriding. These data are compared against the performance achieved by a code optimized without inlining but using a hardware prediction overriding mechanism. The main observation is that, when using aggressive procedure inlining, a processor using the stream fetch engine without overriding is able to outperform a similar processor using code optimized without inlining, even if it uses prediction overriding.

Using a 64KB instruction cache, the processor without overriding executing inlined code only achieves 1.5% reduction in the total number of execution cycles over the processor executing non-inlined code using overriding. Although most improvement is due to the better fetch bandwidth and the additional optimizations enabled, it is interesting to note that the processor without overriding executing inlined code would be unable to outperform the processor with overriding executing non-inlined code if the 2% improvement provided by longer streams is not taken into account. If we could obtain longer streams, maintaining the instruction cache miss rate under control, the execution engine of wide superscalar processors can be feed during multiple cycles by a single stream, hiding the latency of the next stream prediction. This illustrates how the fetch engine complexity can be reduced by exploiting the advantages provided by code optimization.

7.2.2 Individual Benchmark Performance

To achieve a better understanding of our results, we show individual benchmark data for the 8KB instruction cache setup using the corresponding optimal MRCS. We selected this cache because its small size aggravates the problem of instruction cache misses. Figure 19 shows the average length of instruction streams for both the code optimized without inlining and with inlining. There is a clear stream enlargement in five of the ten evaluated benchmarks, specially in *255.vortex*, while the other five benchmarks present little variation. This enlargement is due to an important reduction in the number of taken branches. The five benchmarks that obtain longer streams

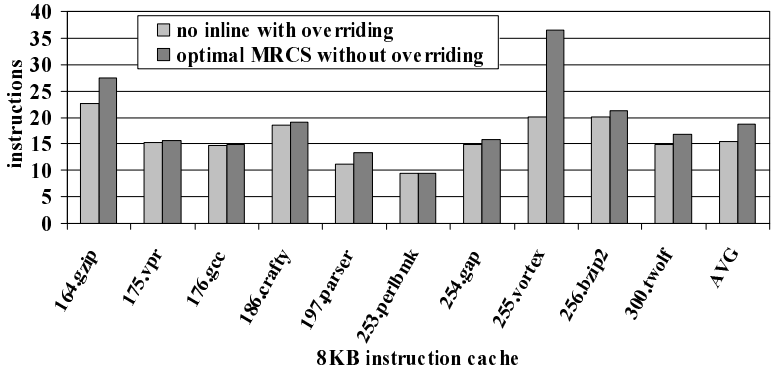


Figure 19. Stream length for individual benchmarks using an 8KB instruction cache, and for both code optimized without inlining and with inlining.

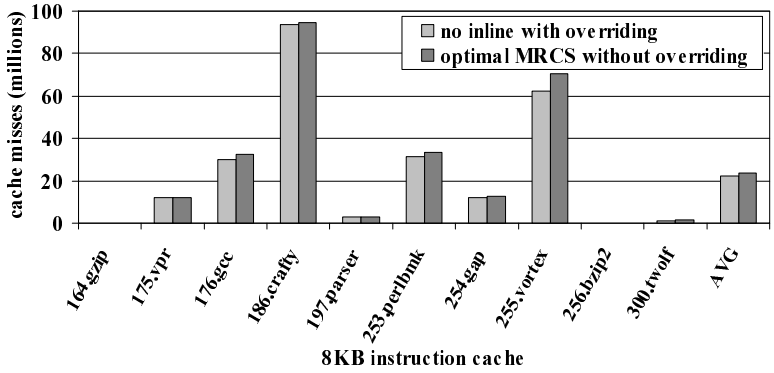


Figure 20. Instruction cache misses for individual benchmarks using an 8KB instruction cache, and for both code optimized without inlining and with inlining.

using inlining achieve a reduction over 20% in the total number of function calls and return instructions, reaching a reduction over 90% in benchmarks like *164.gzip*. Both procedure calls and returns are specially frequent in *255.vortex*, which explains the high increase this benchmark achieves.

As stated before, this enlargement of instruction streams is obtained at the cost of a higher number of instruction cache misses. Figure 20 shows the number of instruction cache misses for the 8KB instruction cache using both the code optimized without inlining and with inlining. Inlining causes an increase in the number of instruction cache misses for *176.gcc*, *253.perlbmk*, and, especially, *255.vortex*. The increase in *255.vortex* is expected due to the great increase in the length of instruction streams. This is the price that must be paid for obtaining so long streams.

Figure 21 shows the performance achieved by the ten benchmarks for both the code optimized without inlining using prediction overriding, and the code optimized with inlining not using overriding. The performance of both processors is close in most benchmarks. The processor without overriding, in spite of being less complex, is even

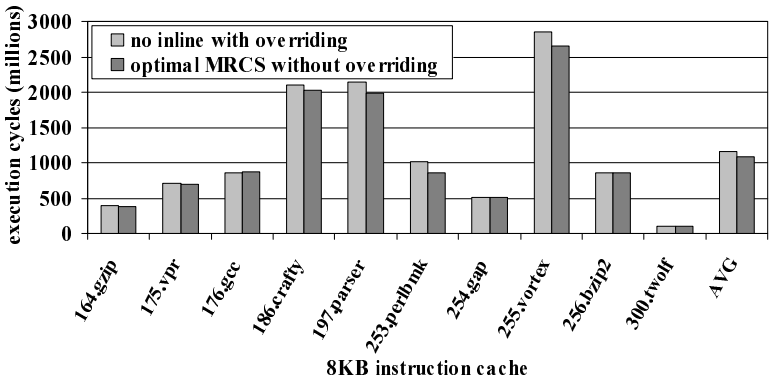


Figure 21. Processor performance for individual benchmarks in an 8-wide processor using an 8KB instruction cache, and for both code optimized without inlining and with inlining.

better in four benchmarks. A clear example is *255.vortex*, which obtains an important reduction in the execution time using aggressive procedure inlining. Although it suffers from a higher number of instruction cache misses, the longer streams obtained in *255.vortex* when using inlining compensate this effect, allowing to achieve a better performance by hiding the prediction table access latency. This shows that software optimization techniques are able to replace a complex hardware mechanism, like prediction overriding, reducing the fetch engine cost and complexity without sacrificing performance.

8 Multiple Stream Prediction

The previously proposed techniques enlarge instruction streams, but they are not able to achieve good performance results. Including short forward branches in streams covers a very small percentage of all dynamic streams, and thus the performance improvement expected from applying this technique is negligible. Loop stream prediction covers a slightly higher percentage of the executed streams, but it tends to enlarge streams that are already long enough, not providing any benefit to the ability of tolerating the prediction table access latency. Finally, aggressive procedure inlining provides better performance results, but the improvement due to longer streams is roughly an average 2%.

The poor results achieved by these techniques show that focusing on particular types of branches is not a correct approach to enlarge instruction streams. Indeed, as described in Section 4, there is a large amount of short streams, having one or few instructions, that are not necessarily enlarged by the aforementioned techniques. Therefore, we must try to enlarge not particular stream types, but all instruction streams. Our approach to achieve this is the multiple stream predictor.

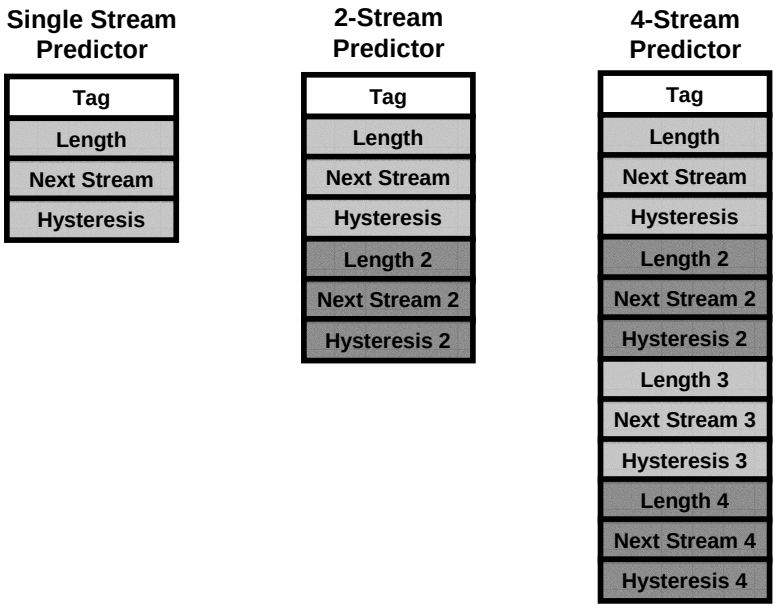


Figure 22. Fields required by the multiple stream predictor for predicting sequences composed by 1, 2, and 4 instruction streams.

8.1 The Multiple Stream Predictor

The objective of our multiple stream predictor is predicting together those streams that are frequently executed as a sequence. Unlike the trace cache, the instructions corresponding to a sequence of streams are not stored together in a special purpose buffer. The instruction streams belonging to a predicted sequence are still separate streams stored in the instruction cache. Therefore, the multiple stream predictor does not enable the ability of fetching instructions beyond a taken branch in a single cycle. The benefit of our technique comes from grouping predictions, allowing to tolerate the prediction table access latency.

Figure 22 shows the fields required by the multiple stream predictor. A single stream predictor, as described in [19, 24], requires the tag field for detecting table hits or misses, the length field to determine what instructions should be executed, the next stream field to decide the next fetch address, and the hysteresis counter for determining when a stream should be replaced from the prediction table. A 2-stream predictor still requires a single tag field, which corresponds to the starting address of the stream sequence. However, the rest of the fields should be duplicated. The tag and length fields determine the first stream that should be executed. The target of this stream, determined by the next stream field, is the starting address of the second stream, whose length is given by the second length field. The second next stream field is the target of the second stream, and thus the next fetch address.

In this way, a single prediction table lookup provides two separate stream predictions, which are supposed to be executed sequentially. Extending this mechanism for predicting four or more streams per sequence is straightforward, as shown in Figure 22. After a multiple stream prediction, every stream belonging to a predicted sequence is stored separately in the FTQ, which involves that using the multiple-stream predictor does not require additional changes in the processor front-end.

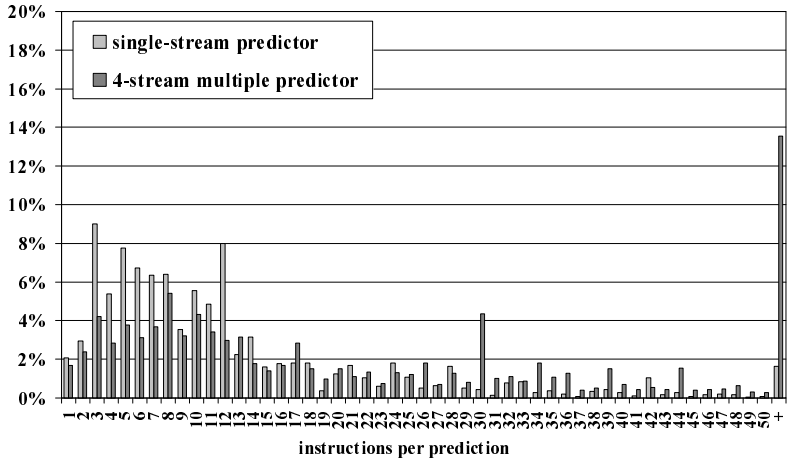
In order to detect frequently executed stream sequences, we use the hysteresis counters. Every stream in a sequence has a hysteresis counter associated to it. All hysteresis counters behave like the counter used by the original stream predictor to decide whether a stream should be replaced from the prediction table [24]. When the predictor is updated with a new stream, the corresponding counter is increased if the new stream matches with the stream already stored in the selected entry. Otherwise, the counter is decreased and, if it reaches zero, the whole predictor entry is replaced with the new data, setting the counter to one. If the decreased counter does not reach zero, the new data is discarded.

When the prediction table is looked up, the first stream is always provided. The second stream is only predicted if the corresponding hysteresis counter is saturated, that is, if the counter has reached its maximum value. The third stream is just predicted if both the second and third hysteresis counters are saturated, and so on. Therefore, if no hysteresis counter is saturated, the multiple stream predictor provides a single stream prediction as it would be done by the original stream predictor. On the contrary, if several consecutive hysteresis counters have saturated, then a frequently executed sequence has been detected, and all streams belonging to this sequence are introduced in the FTQ.

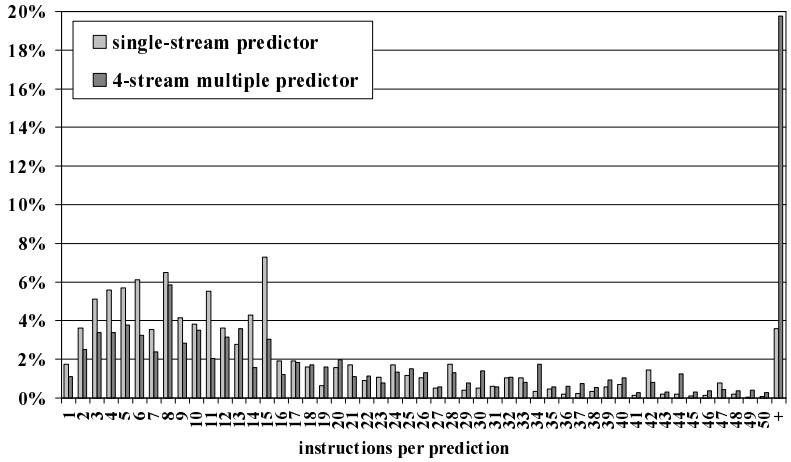
The drawback of this technique is that providing multiple streams per prediction needs an important increase in the prediction table size. In order to avoid a negative impact on the prediction table access latency, we only store multiple streams in the first-level table of the cascaded stream predictor, which is smaller than the second-level table. Since the streams belonging to a sequence are supposed to be frequently executed together, it is likely that, given a fetch address, the executed sequence is always the same. Consequently, stream sequences do not need correlation to be correctly predicted, and thus keeping them in the first level table does not limit the achievable benefit.

8.2 Multiple Stream Prediction Evaluation

Our multiple stream predictor is able to provide a high amount of instructions per prediction. Figure 23 shows an histogram of instructions provided per prediction. It shows the percentage of predictions that provided an amount of instructions ranging from 1 to 50 instructions. The last bar shows the percentage of predictions that provided more than 50 instructions. Data are shown for both the baseline and the optimized code layout. In addition, data are shown for the original single-stream predictor, described in [24], and a 4-stream multiple predictor.



(a) baseline code



(b) optimized code

Figure 23. Histograms of dynamic predictions, classified according to the amount of instructions provided, when using a single-stream predictor and a 4-stream multiple predictor. The results presented in these histograms are the average of the eleven benchmarks used.

It is clear that our multiple stream predictor efficiently deals with the most harmful problem, that is, the shorter streams. There is an important reduction in the percentage of predictions that provide a small number of instructions, especially when using optimized codes. Furthermore, there is an impressive increase in the percentage of predictions that provide more than 50 instructions. These results show that the multiple stream predictor is an effective technique for hiding the prediction table access latency by overlapping table accesses with the execution of useful instructions.

Figure 24 shows the average processor performance achieved by the four evaluated fetch architectures, for both the baseline and the optimized code layout. We have evaluated a wide range of predictor setups and selected the best one found for each evaluated predictor. Besides the performance of the four fetch engines using overriding,

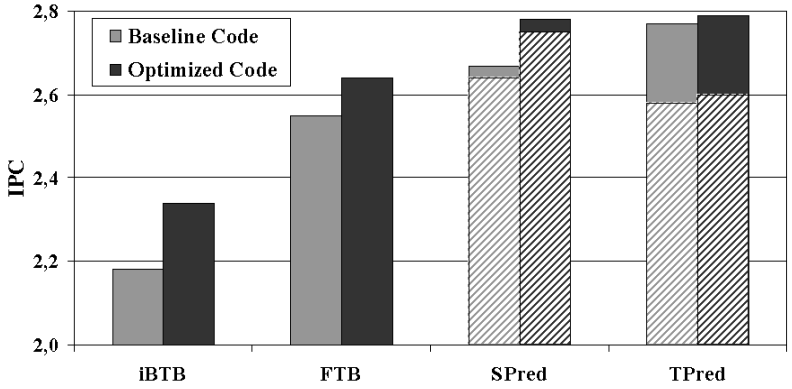


Figure 24. Processor performance when using (full bar) and not using (shadowed bar) overriding.

the performance achieved by the trace cache fetch architecture and the stream fetch engine not using overriding is also shown. In the latter case, the stream fetch engine uses a 4-stream multiple predictor instead of the original single-stream predictor.

The main observation from Figure 24 is that the multiple stream predictor without overriding provides a performance very close to the original single-stream predictor using overriding. The performance achieved by the multiple stream predictor without overriding in enough to outperform both the iBTB and the FTB fetch architectures, even when they do use overriding. The performance of the multiple stream predictor without overriding is also close to a trace cache using overriding, while requiring lower complexity.

However, using a 4-stream multiple predictor involves increasing the first-level table size by almost a factor of four. As a consequence, the 4-stream multiple predictor requires 40% more hardware budget than the original single-stream predictor. Although we have measured using CACTI [29] that this increase in the predictor size has not a negative impact on the predictor access latency, it causes an average 50% increase in the overall predictor energy consumption. Nevertheless, the multiple stream predictor does not require a complex overriding mechanism for overcoming the problem of access latency, which will also involve an increase in the energy consumption. Our proposal requires larger prediction tables, but lower overall complexity, which would be an interesting alternative for future processor designers.

9 Conclusions

Current technology trends create new challenges for the fetch architecture design. Higher clock frequencies and larger wire delays cause branch prediction tables to require multiple cycles to be accessed [1, 13], limiting the fetch engine performance. This fact has led to the development of complex hardware mechanisms, like prediction overriding [13, 26], to hide the prediction table access delay.

To avoid this increase in the fetch engine complexity, we propose to use long instruction streams for hiding the prediction table access delay. If instruction streams are long enough, the execution engine can be kept busy executing instructions from a stream during multiple cycles, while a new stream prediction is being generated. Therefore, the prediction table access delay can be hidden without requiring any additional hardware mechanism.

In order to take maximum advantage of this fact, it is important to have streams as long as possible. We have presented a detailed analysis of dynamic instruction streams, showing that most of them finalize in conditional branches, function calls, and return instructions. Thus, we consider these types of instructions are the best candidates for enlarging instruction streams. However, our first attempts to enlarge instruction streams have not provided good results.

We have presented a technique for including short forward branches in instruction streams, but it is applicable to a little percentage of existing branches, and thus it would have negligible impact on performance. We have also presented a loop stream predictor, which enlarges streams by combining loop iterations, but it tends to enlarge streams that are already long enough for tolerating access latency. Regarding function calls and returns, we have shown that aggressive procedure inlining removes a high percentage of these taken branches, enlarging streams. However, most benefit provided by this technique is not due to the longer size of streams, but to the improved fetch bandwidth and the additional optimizations enabled.

Therefore, we conclude that focusing on particular branch types is not the correct approach for enlarging instruction streams. Using the techniques mentioned above, we can achieve long streams, but there are also other short streams that limit the achievable benefit, according to Amdahl's law. In order to overcome this problem, we propose a multiple stream predictor that combines frequently executed streams into long virtual streams. This new predictor does not take into account the type of the branch finalizing the stream, and thus it is a more general approach. As a consequence, our novel predictor design provides streams long enough for allowing a processor not using overriding to achieve a performance very close to a processor using prediction overriding.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIN–2004–07739–C02–01, the HiPEAC European Network of Excellence, CEPBA, and an Intel fellowship.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[2] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1988.

[3] H. Aydin and D. Kaeli. Using cache line coloring to perform aggressive procedure inlining. *ACM Computer Architecture News*, 28(1), 2000.

[4] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1997.

[5] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[6] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: an optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*, 1997.

[7] A. Falcon, O. J. Santana, A. Ramirez, and M. Valero. Tolerating branch predictor latency on SMT. In *Proceedings of the 5th International Symposium on High Performance Computing*, 2003.

[8] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), 1996.

[9] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal useful logic depth per pipeline stage is 6-8 fo4. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[10] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1989.

[11] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[12] D. A. Jimenez. Reconsidering complex branch predictors. In *Proceedings of the 9th International Conference on High Performance Computer Architecture*, 2003.

[13] D. A. Jimenez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[14] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, 2001.

[15] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th International Symposium on Computer Architecture*, 1991.

[16] A. KleinOsowski and D. J. Lilja. MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research. *IEEE TCCA Computer Architecture Letters*, 1, 2002.

[17] R. Muth, S. K. Debray, S. A. Watterson, and K. D. Bosschere. ALTO: a link-time optimizer for the Compaq Alpha. *Software - Practice and Experience*, 31(1), 2001.

[18] A. Ramirez, J. L. Larriba-Pey, and M. Valero. Trace cache redundancy: red & blue traces. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, 2000.

[19] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. In *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.

[20] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[21] R. Rosner, A. Mendelson, and R. Ronen. Filtering techniques to improve trace cache efficiency. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[22] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2), 1999.

[23] O. J. Santana, A. Falcón, E. Fernández, P. Medina, A. Ramirez, and M. Valero. A comprehensive analysis of indirect branch prediction. *Proceedings of the 4th International Symposium on High Performance Computing*, 2002.

[24] O. J. Santana, A. Ramirez, J. L. Larriba-Pey, and M. Valero. A low-complexity fetch architecture for high-performance superscalar processors. *ACM Transactions on Architecture and Code Optimization*, 1(2), 2004.

[25] O. J. Santana, A. Ramirez, and M. Valero. Latency tolerant branch predictors. In *Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2003.

[26] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[27] A. Seznec and A. Fraboulet. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th International Symposium on Computer Architecture*, 2003.

[28] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[29] P. Shivakumar and N. P. Jouppi. CACTI 3.0: an integrated cache timing, power and area model. Technical Report 2001/2, Western Research Laboratory, 2001.

[30] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.