Predicting Multiple Streams per Cycle

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

The next stream predictor is an accurate branch predictor that provides stream level sequencing. Every stream prediction contains a full stream of instructions, that is, 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 the stream predictor to provide high fetch bandwidth and to tolerate the prediction table access latency. Therefore, an excellent way for improving the behavior of the next stream predictor is to enlarge instruction streams.

In this paper, 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 table access latency without requiring the complexity caused by additional hardware mechanisms like prediction overriding, also reducing the overall branch predictor energy consumption.

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 [12, 17].

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

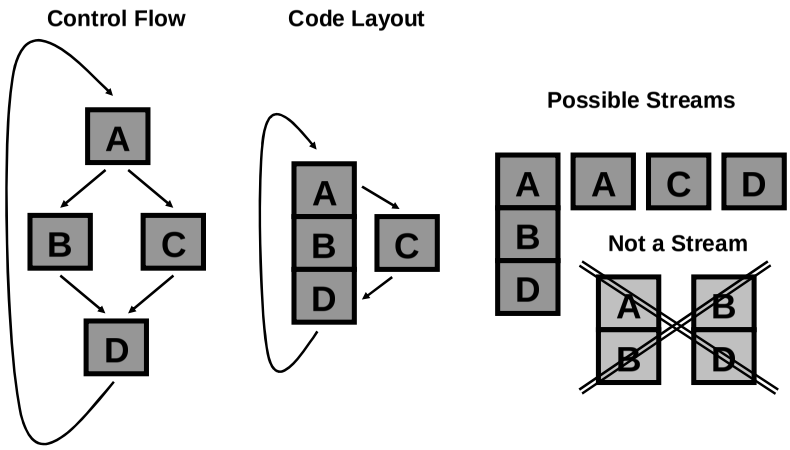


Figure 1. Example of instruction streams.

structure. Let us suppose that 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, 8]. 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 [8, 20]. 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 [17]. 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 [17]. These optimizations try to map together those basic blocks that 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 [18].

Looking for novel ways of enlarging streams, we present a detailed analysis of dynamic instruction streams. Our results show that most of them finalize in conditional branches, function calls, and return instructions. As a consequence, it would seem that these types of branches are the best candidates to apply techniques for enlarging instruction streams. However, according to Amdahl's law, focusing on particular branch types is not a good approach to enlarge instruction streams. If we focus on a particular type of stream, the remainder streams, which do not benefit from the stream enlargement, will 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 the multiple stream predictor. Section 6 evaluates the proposed predictor. Finally, Section 7 presents our concluding remarks.

2 Related Work

The prediction table access latency is 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, 8].

The trace predictor [6] 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 [18], 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) [13] 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.

Another promising idea to tolerate the prediction table access latency is pipelining the branch predictor [7, 21]. Using a pipelined predictor, a new prediction can be started each cycle. Nevertheless, this is not trivial, since the outcome 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 [21], 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 [24] by pipelining the branch predictor and interleaving prediction requests from different threads each cycle [3]. 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. [8]. 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 [4] and EV8 [20] processors, where a multi-cycle latency branch predictor overrides a faster but less accurate cache line predictor [2].

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 [22].

Since previous work [17] 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.

[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.

fetch, rename, and commit widths	8 instructions
integer and floating point issue widths	8 instructions
load/store issue width	4 instructions
fetch target queue	8 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	64/32 KB, 2-way associative, 128 byte block, 3 cycle latency
L1 data cache	64 KB, 2-way associative, 64 byte block, 3 cycle latency
L2 unified cache	1 MB, 4-way associative, 128 byte block, 16 cycle latency
main memory latency	350 cycles
maximum trace size	32 instructions (10 branches)
filter and main trace caches	128 traces, 4-way associative

Table 1. Configuration of the simulated processor

3.1 Simulator Setup

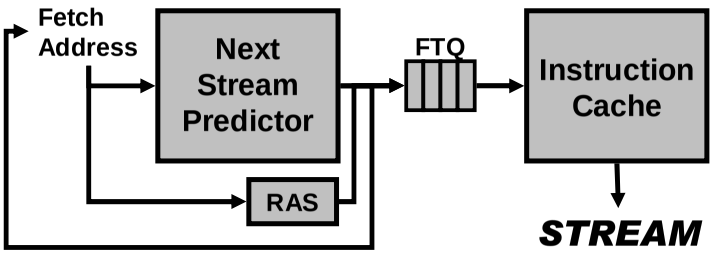
Our simulation setup corresponds to an aggressive 8-wide superscalar processor. The main values of this setup are shown in Table 1. We compare our stream fetch architecture with three other state-of-the-art fetch architectures: a fetch architecture using an interleaved BTB and a 2bcgskew predictor [20], the fetch target buffer (FTB) architecture [13] using a perceptron predictor [9], and the trace cache fetch architecture using a trace predictor [6]. All these architectures use an 8-entry fetch target queue (FTQ) [13] 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 [12], and 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 [14] and a main trace cache. In addition, we use selective trace storage [11] to avoid trace redundancy between the trace cache and the instruction cache.

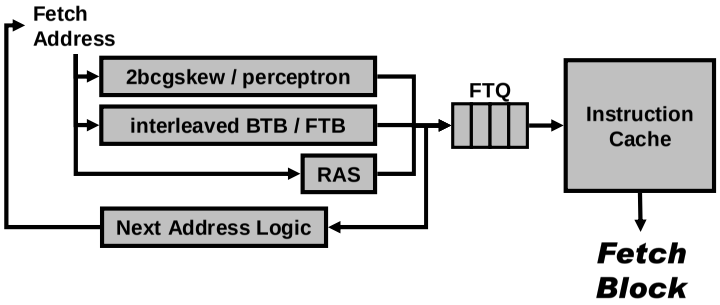
3.2 Fetch Models

The stream fetch engine [12, 17] 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 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.

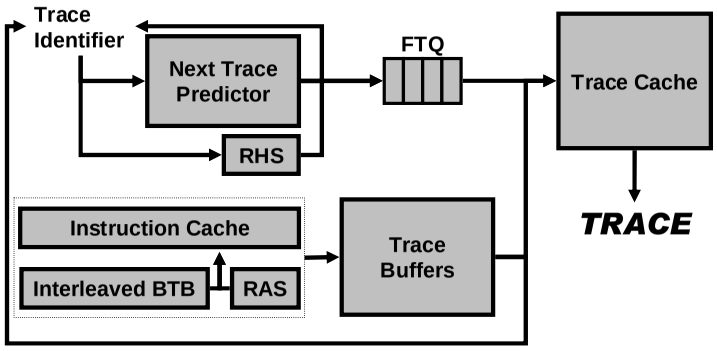
Our interleaved BTB fetch model (iBTB) is inspired by the EV8 fetch engine design described in [20]. This iBTB model decouples the branch prediction mechanism from the instruction cache with an FTQ. An interleaved BTB is used to allow the prediction of multiple branches until a taken branch is predicted, or until an aligned 8-instruction block is completed. The branch prediction history is updated using a single bit for prediction block,



(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.

which combines the outcome of the last branch in the block with path information [20]. Our FTB model is similar to the one described in [13] but using a perceptron branch predictor [9] 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 [15] but enhanced using an FTQ [13] 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) [10] 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 [16] 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) [6] 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.

3.3 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 [23]. 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 [5] 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 [18]. 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 achieve poor performance. Therefore, we have chosen to simulate all branch predictors using the bigger tables that can be accessed in three cycles. Table 2 shows the configuration of the simulated predictors. We have explored a wide range of history lengths, as well as DOLC index [6] configurations, and selected the best one found for each setup. Table 2 also shows the approximated hardware budget for each predictor. Since we simulate the larger three cycle latency tables[2], 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.

[2]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.

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 2. Configuration of the simulated branch predictors.

Our fetch models also use an overriding mechanism [8, 20] 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 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 2.

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 requiring low implementation cost.

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 [17], 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

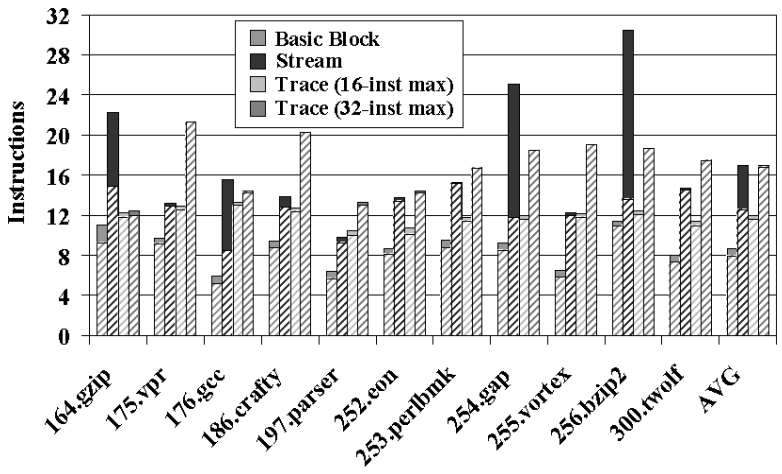


Figure 3. Average length of basic blocks, instruction streams, and instruction traces for both baseline codes (shadowed bar) and optimized codes (full bar).

stored in a special purpose cache, their size is physically limited. Using a maximum trace size of 16 instructions [15] 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 [17], 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. On the contrary, 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 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 that is responsible for 54% of the total number of streams.

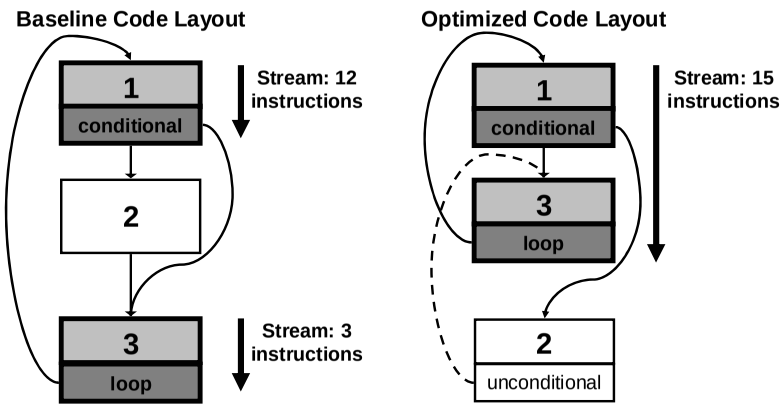


Figure 4. Example of code layout optimization taken from the *176.gcc* benchmark.

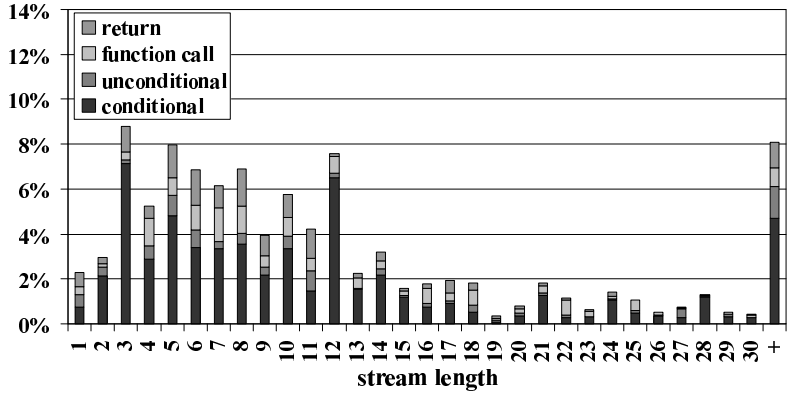
The increase in the average stream length achieved by using optimized codes is beneficial for the next stream predictor accuracy. Having longer streams causes most part of the program execution to be held in a lower number of streams. This fact reduces aliasing in the prediction table, increasing prediction 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

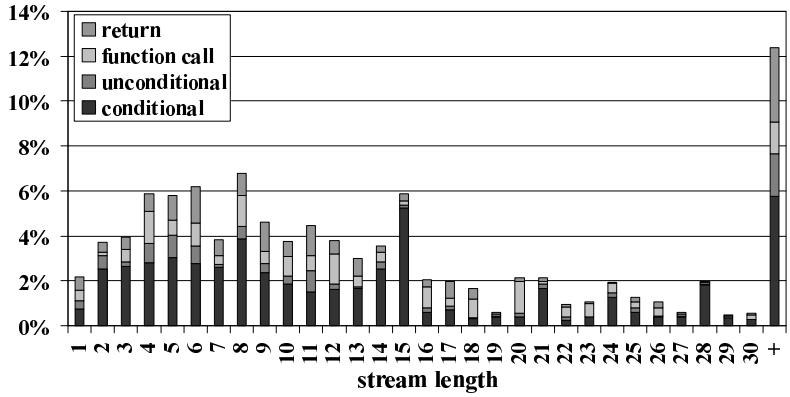
Longer streams make it possible for the stream fetch engine to achieve better performance. However, having high average length does not involve that most streams are long. Some streams could be long, providing high fetch bandwidth, while other streams could be short, limiting the potential performance.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

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 longer 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 higher. However, most streams are still shorter than the average length: 70% of the dynamic streams have 15 or less instructions. Therefore, in



(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.

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.

5 Multiple Stream Prediction

According to the analysis presented in the previous section, one could think that, in order to enlarge instruction streams, the most promising field for research are conditional branches, function calls, and return instructions. However, we have found that techniques for enlarging the streams finalizing in particular branch types achieve poor results [19]. This is due to Amdahl's law: although these techniques enlarge a set of instructions streams, there are other streams that are not enlarged, limiting the achievable benefit. Therefore, we must try to enlarge not particular stream types, but all instruction streams. Our approach to achieve this is the multiple stream predictor.

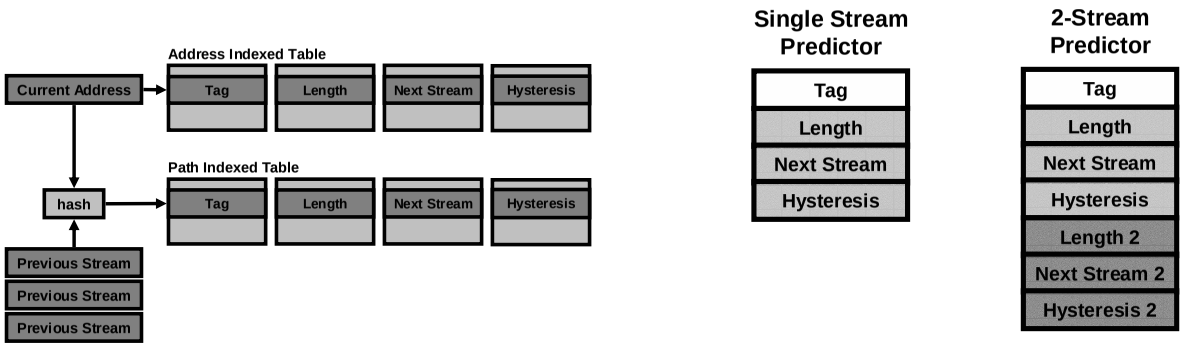
5.1 The Multiple Stream Predictor

The next stream predictor [12, 17], which is shown in Figure 6.a, is a specialized branch predictor that provides stream level sequencing. Given a fetch address, i.e., the current stream starting address, the stream predictor provides the current stream length, which indicates where is the taken branch that finalizes the stream. The predictor also provides the next stream starting address, which is used as the fetch address for the next cycle. The current stream starting address and the current stream length form a fetch request that is stored in the FTQ. The fetch requests stored in the FTQ are then used to drive the instruction cache.

Actually, the stream predictor is composed by two cascaded tables: a first level table indexed only by the fetch address, and a second level table indexed using path correlation. A stream is only introduced in the second level if it is not accurately predicted by the first level. Therefore, those streams that do not need correlation are kept in the first level, avoiding unnecessary aliasing. In order to generate a prediction, both levels are looked up in parallel. If there is a second level table hit, its prediction is used. Otherwise, the prediction of the first level table is used. The second level prediction is prioritized because it is supposed to be more accurate than the first level due to the use of path correlation.

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 6.b shows the fields required by a 2-stream multiple predictor. Like the original single stream predictor, a 2-stream predictor 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



(a) cascaded predictor design (b) fields required for multiple stream prediction

Figure 6. The next stream predictor.

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. 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. Extending this mechanism for predicting three or more streams per sequence would be straightforward, but we have found that sequences having more than two streams do not provide additional benefit.

5.2 Multiple Stream Predictor Design

Providing two streams per prediction needs duplicating the prediction table size. In order to avoid a negative impact on the prediction table access latency and energy consumption, 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.

In order to take maximum advantage of the available space in the first level table, we use hysteresis counters to detect frequently executed stream sequences. 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 [17]. 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. We have found

that 3-bit hysteresis counters, increased by one and decreased by two, provide the best results for the multiple stream predictor.

When the prediction table is looked up, the first stream is always provided. However, the second stream is only predicted if the corresponding hysteresis counter is saturated, that is, if the counter has reached its maximum value. Therefore, if the second hysteresis counter is not saturated, the multiple stream predictor provides a single stream prediction as it would be done by the original stream predictor. On the contrary, if the two hysteresis counters are saturated, then a frequently executed sequence has been detected, and the two streams belonging to this sequence are introduced in the FTQ.

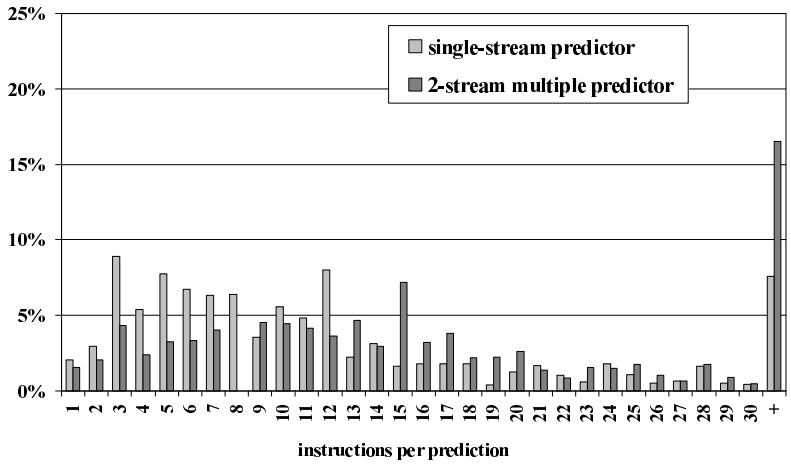
6 Evaluation of the Multiple Stream Prediction

Our multiple stream predictor is able to provide a high amount of instructions per prediction. Figure 7 shows an histogram of instructions provided per prediction. It shows the percentage of predictions that provide an amount of instructions ranging from 1 to 30 instructions. The last bar shows the percentage of predictions that provide more than 30 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 [17], and a 2-stream multiple predictor.

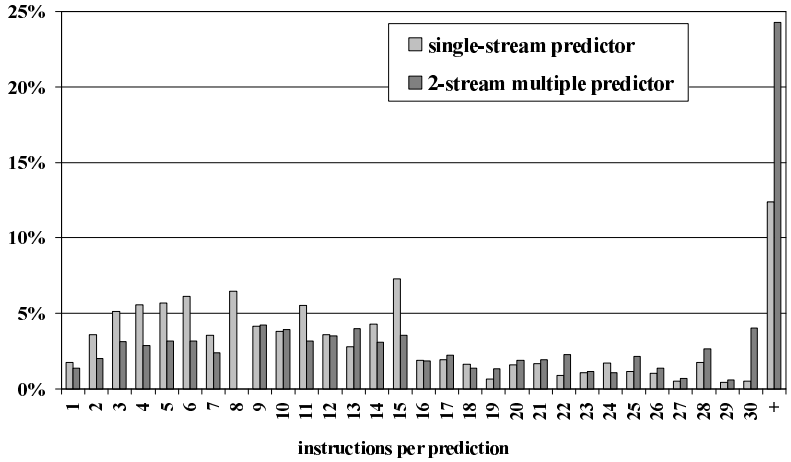
The main difference between both code layouts is that, as can be expected, there is a lower percentage of short streams in the optimized code. Besides, it is clear that our multiple stream predictor efficiently deals with the most harmful problem, that is, the shorter streams. Using our multiple stream predictor, there is an important reduction in the percentage of predictions that provide a small number of instructions. Furthermore, there is an increase in the percentage of predictions that provide more than 30 instructions, especially when using optimized codes. The lower number of short streams points out 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 8 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, 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 2-stream multiple predictor instead of the original single-stream predictor.

The main observation from Figure 8 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.



(a) baseline code



(b) optimized code

Figure 7. Histograms of dynamic predictions, classified according to the amount of instructions provided, when using a single-stream predictor and a 2-stream multiple predictor. The results presented in these histograms are the average of the eleven benchmarks used.

Finally, it should be taken into account that this improvement is achieved by increasing the size of the first level table. Fortunately, the tag array is unmodified and no additional access port is required. We have checked using CACTI [23] that the increase in the predictor area is less than 12%, as well as that the prediction table access latency is not increased. Moreover, our proposal not only avoids the need for a complex overriding mechanism, but also reduces the predictor energy consumption. Although the bigger first level table consumes more energy per access, it is compensated with the reduction in the number of prediction table accesses. The ability of providing two streams per prediction causes a 35% reduction in the total number of prediction table lookups an updates, which leads to a 12% reduction in the overall stream predictor energy consumption.

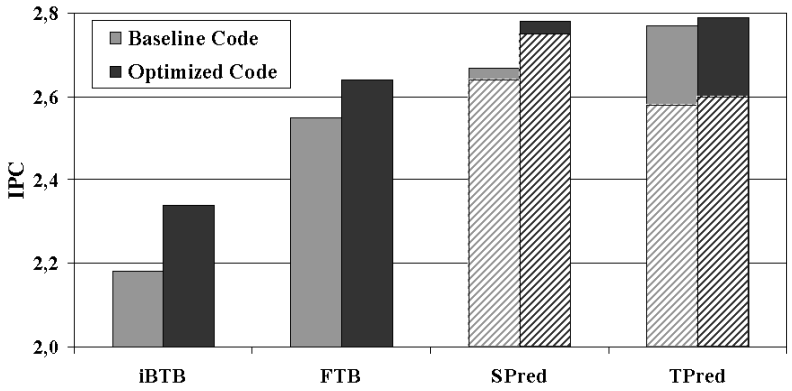


Figure 8. Processor performance when using (full bar) and not using (shadowed bar) overriding.

7 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, 8], limiting the fetch engine performance. This fact has led to the development of complex hardware mechanisms, like prediction overriding [8, 20], to hide the prediction table access delay.

To avoid this increase in the fetch engine complexity, we propose to use long instruction streams [12, 17] as basic prediction unit, which makes it possible to hide 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 achieve this using the multiple stream predictor, a novel predictor design that combines frequently executed instruction streams into long virtual streams. Our predictor provides instruction streams long enough for allowing a processor not using overriding to achieve a performance close to a processor using prediction overriding, that is, we achieve a very similar performance at a much lower complexity, also requiring less energy consumption.

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

[3] 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.

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

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] O. J. Santana, A. Ramirez, and M. Valero. Techniques for enlarging instruction streams. Technical Report UPC-DAC-RR-2005-11, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, 2005.

[20] 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.

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

[22] 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.

[23] 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.

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