Enlarging Instruction Streams

Oliverio J. Santana, *Member*, *IEEE*, Alex Ramirez, *Member*, *IEEE*, and Mateo Valero, *Fellow*, *IEEE*

Abstract—The stream fetch engine is a high-performance fetch architecture based on the concept of an instruction stream. We call a sequence of instructions from the target of a taken branch to the next taken branch, potentially containing multiple basic blocks, a stream. The long length of instruction streams makes it possible for the stream fetch engine to provide a high fetch bandwidth and to hide the branch predictor access latency, leading to performance results close to a trace cache at a lower implementation cost and complexity. Therefore, enlarging instruction streams is an excellent way to improve the stream fetch engine. In this paper, we present several hardware and software mechanisms focused on enlarging those streams that finalize at particular branch types. However, our results point out that focusing on particular branch types is not a good strategy due to Amdahl's law. Consequently, we propose the multiple-stream predictor, a novel mechanism that deals with all branch types by combining single streams into long virtual streams. This proposal tolerates the prediction table access latency without requiring the complexity caused by additional hardware mechanisms like prediction overriding. Moreover, it provides high-performance results which are comparable to state-of-the-art fetch architectures but with a simpler design that consumes less energy.

Index Terms—Superscalar processor design, instruction fetch, branch prediction, access latency, code optimization.

1 Introduction

High-performance superscalar processors require efficient fetch architectures to exploit all of 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 achieving high fetch performance while maintaining the complexity under control is to use the stream fetch engine [1], [2].

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 a sequence of instructions from the target of a taken branch to the next taken branch, potentially containing multiple basic blocks, a stream. Fig. 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 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 path $A \rightarrow B$ goes through a not-taken branch and falls through to 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 sequence AB nor 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 is a three-stream sequence that follows the taken branch at the end of A, goes through C, and jumps back into basic block D.

The stream fetch engine provides accurate predictions that contain enough instructions to provide a high fetch bandwidth. However, taking into account current technology trends, accurate branch prediction and high fetch bandwidth are not enough. The continuous increase in processor clock frequency, as well as the larger wire delays caused by modern technologies, prevents branch prediction tables from being accessed in a single cycle [3], [4]. This fact 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 [4], [5]. 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 [6]. 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 while a new prediction is being generated. Overlapping the execution of a prediction with the generation of the following prediction allows

- *O.J. Santana is with the Departamento de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, Campus Universitario de Tafira, 35017 Las Palmas de Gran Canaria, Spain. E-mail: ojsantana@dis.ulpgc.es.*
- *A. Ramirez and M. Valero are with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, and the Barcelona Supercomputing Center, Campus Nord UPC, Jordi Girona 1-3, 08034 Barcelona, Spain. E-mail: {aramirez, mateo}@ac.upc.edu.*

Manuscript received 28 Apr. 2006; revised 5 Dec. 2006; accepted 26 Apr. 2007; published online 6 June 2007.
Recommended for acceptance by M. Dubois.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0162-0406.
Digital Object Identifier no. 10.1109/TC.2007.70742.

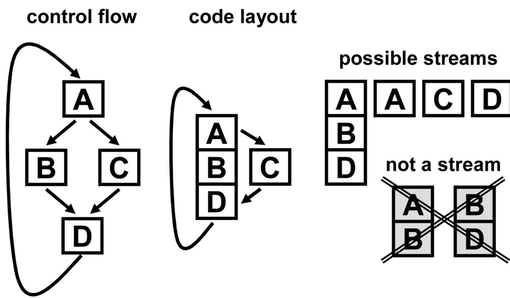

Fig. 1. Example of instruction streams.

partially hiding 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 by removing taken branches. Code layout optimizations try to map together those basic blocks that are frequently executed as a sequence. Therefore, most conditional branches in optimized codes are not taken, enlarging instruction streams [1], [2]. However, code layout optimizations are not enough for the stream fetch engine to completely overcome the need for an overriding mechanism [6].

Looking for novel ways of enlarging streams, we present several mechanisms focused on enlarging the instruction streams that finalize in particular branch types. All of these mechanisms have a similar objective: reducing the total amount of taken branches of a certain type and thus enlarging instruction streams.

We introduce two new versions of the next-stream predictor aimed at removing taken conditional branches. 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. For removing backward conditional branches, we present a loop-stream predictor, that is, a version of the stream predictor that is able to predict high-level loop structures, making it possible to combine all of the iterations of simple loops into a single long virtual stream. Finally, in order to remove function calls and returns, we apply aggressive procedure inlining [7] to replace procedure calls by the called procedures.

Unfortunately, these techniques do not provide good performance results since they are not enlarging all streams but just a subset of them. According to Amdahl's law, the streams that do not benefit from the previous techniques severely limit the potential performance improvement they can achieve. This fact leads to a clear conclusion: The correct approach is not to focus on particular branch types but to try to enlarge all dynamic streams. In order to achieve this, we present the multiple-stream predictor [8], 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.

Our results show that the multiple-stream predictor provides long branch predictions with high accuracy, allowing our proposal to achieve performance results comparable to state-of-the-art fetch architectures. Furthermore, our predictor does not require hardware overriding to hide the prediction table access latency and, thus, it requires less implementation cost and complexity than previous proposals.

The remainder of this paper is organized as follows: Section 2 describes previous related work. Section 3 presents our experimental methodology. Section 4 describes a technique for including short forward branches (SFBs) in instruction streams. Section 5 presents a novel loop-stream predictor. Section 6 details the impact of aggressive procedure inlining on the length of instruction streams. Section 7 describes the multiple-stream predictor. Finally, Section 8 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 a fetch address in a single cycle because this address is needed for fetching instructions in the next cycle. However, the continuous increase in processor clock frequency, as well as the slower wires in modern technologies, causes branch prediction tables to require multicycle accesses [3], [4].

A straightforward technique for overcoming this problem is by using long basic prediction units. If a branch prediction provides enough instructions, they will keep the processor back end busy during the time required to generate the next prediction, effectively hiding the branch predictor access delay.

The trace predictor [9] is an efficient mechanism for obtaining long basic prediction units since each trace prediction is potentially a multiple-branch prediction. The SEQ.n [10] fetch model defines a trace as a sequence of basic blocks separated by not-taken branches. A trace predictor is used to address an interleaved instruction cache and to obtain instruction sequences that are potentially long enough to feed the processor back end during multiple cycles. This trace definition can be extended to any number of nonsequential basic blocks, enlarging the basic prediction unit at the cost of increasing the fetch architecture complexity due to the need for a special-purpose trace cache [11] to store together the nonsequential instructions.

Our instruction streams are conceptually similar to the basic prediction unit defined by the SEQ.n model: dynamic sequences of basic blocks containing just not-taken branches. The main difference between these two approaches is that the SEQ.n model predicts each basic block in a sequence, whereas the stream fetch engine predicts the sequence as a whole. Consequently, the stream fetch engine requires less data to generate predictions and, thus, it requires less implementation complexity. Overall, instruction streams are not as long as traces containing multiple nonsequential basic blocks [1], [2]. However, they are long enough to partially hide the prediction table access latency [6].

Using a fetch target queue (FTQ) [12] is helpful for taking advantage of long basic prediction units. The FTQ decouples the branch prediction mechanism and the instruction cache access. In each cycle, the branch predictor generates the fetch address for the next prediction 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 one hand, code optimizations can be used to rearrange the code and to increase the probability of overlapping branch predictor accesses with the execution of useful work. On the other hand, hardware mechanisms like prediction overriding can be used to reduce 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 previous prediction, it is important to increase the amount of instructions provided by each prediction, that is, to enlarge 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 change the static mapping of basic blocks, making it possible to layout together those basic blocks that are frequently executed as a sequence. This technique leads to a code mapping where most conditional branches are not taken. Therefore, using code optimizers like Spike [13] allows increasing the length of instruction streams and thus increasing the stream predictor ability to tolerate the access latency.

Code replication optimizations can also be used to enlarge streams by removing taken branches from the critical path of the program execution. Trace scheduling [14] is a technique that selects a sequence of frequently executed basic blocks, with multiple entry and exit points, and schedules the instructions to increase the available instruction-level parallelism. Loop unrolling and static branch prediction allow enlarging these traces by replicating the frequent part of the code. In addition, the compensation code may also be required to deal with the nonfrequent cases. Superblock scheduling [15] is similar to trace scheduling, but, while traces may have several entry points, superblocks must have a single entry point. Tail duplication is used to create a copy of the superblock from any entry point after the first one, effectively creating a separate sequence of instructions.

The main limitation of trace and superblock scheduling is that they contain instructions from a single control path. Hyperblock scheduling [16] overcomes this limitation by using predicated execution [17] to allow the grouping of basic blocks from different control paths. Like superblocks, hyperblocks must have a single entry point and, thus, they require tail duplication to separate additional entries. Loop-unrolling and loop-peeling techniques may also be used to create larger and more efficient hyperblocks.

Procedure inlining is another code replication optimization. This optimization replaces a procedure call by the procedure itself, removing the call and return instructions. In general, procedure inlining is a frequently used code optimization. Allen and Johnson [18] 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 [19] present profile-driven algorithms for applying inlining and code reordering. Profile information is used to decide whether inlining a procedure will be beneficial for the program execution, allowing the inlining of bigger procedures. In addition, reordering the program code is an effective technique to alleviate the increase in the number of instruction cache misses caused by inlining big procedures.

Ayers et al. [20] describe an aggressive inliner, based on profile information, which is able to inline procedures without restriction at almost any call site. Their results show that aggressive inlining can provide important performance improvements in some benchmarks. Likewise, the ALTO [21] optimizer is able to aggressively inline procedures, using profile-based code reordering to reduce the negative effects of inlining on the instruction cache performance. Aydin and Kaeli [22] take this technique one step further by 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.

Although instruction streams may be enlarged using any of these techniques, we mainly focus on code layout optimizations. We have chosen them because they do not involve a direct increase in the code size, which would have a negative impact on the instruction cache performance. As an example of the code replication techniques, we analyze the impact of procedure inlining on the length of instruction streams, showing that longer streams can be obtained at the cost of increasing the total number of instruction cache misses. It would be interesting future work to examine the impact of other code replication techniques, especially predication-based ones, on the length of instruction streams.

2.2 Hardware Mechanisms for Tolerating Access Latency

A promising idea to tolerating the prediction table access latency is to pipeline the branch predictor [23], [24]. Using a pipelined predictor, a new prediction can be started in 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 [24], 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 [25] by pipelining the branch predictor and interleaving prediction requests from different threads [26]. Nevertheless, analyzing the accuracy and performance of pipelined branch predictors is out of the scope of this work.

A different approach is the use of the overriding mechanism described by Jimenez et al. [4]. 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 more accurate. A similar mechanism is used in the Alpha EV6 [27] and EV8 [5] processor designs, where a multicycle latency branch predictor overrides a faster but less accurate cache line predictor [28].

The problem of prediction overriding is that it involves a significant increase in the fetch engine complexity. An overriding mechanism requires a fast branch predictor to

TABLE 1
Configuration of the Simulated Processor

fetch/rename/commit width	8 instructions
int/fp issue width	8 instructions
load/store issue width	4 instructions
fetch target queue	8 entries
instruction fetch queue	32 entries
int/fp issue queues	64 entries
load/store issue queue	64 entries
reorder buffer	256 entries
integer/fp registers	160
L1 instruction cache	64/32KB, 2-way, 128B-block, 3 cycles
L1 data cache	64KB, 2-way, 64B-block, 3 cycles
L2 unified cache	1MB, 4-way, 128B-block, 16 cycles
main memory latency	350 cycles
maximum trace size	32 instructions (10 branches)
filter and main trace caches	128 traces, 4-way

obtain a prediction in each cycle. This prediction should be stored for comparison with the main prediction. Some cycles later, when the main prediction is generated, the fetch engine should determine whether the first prediction is the same or not. If the first prediction is not equal to the main prediction, all of 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 discarded first prediction does not involve discarding all of the instructions fetched from it. Since both the first and the main predictions start at 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 of the complexity of the overriding mechanism. Therefore, the challenge is to develop a technique that is able to tolerate the predictor access latency and to achieve the same performance as an overriding mechanism while 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 an execution-driven simulation of a superscalar processor. Our cycle-accurate simulation tool models a 10-stage processor pipeline. This processor model is able to follow speculative execution paths guided by the branch predictor, as well as to correctly recover the processor state in case a misprediction is detected.

The first four stages constitute the in-order processor front end: prediction, fetch, decode, and rename. The branch predictor is a fully autonomous engine able to generate predictions without further assistance. These predictions guide the fetch engine, providing the data required to drive the instruction cache and obtain instructions from the memory. The fetched instructions are decoded by the corresponding logic and later renamed to resolve data dependencies.

TABLE 2
Simulated Benchmarks and the Corresponding Input Set

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

The fifth stage, dispatch, is the interface between the front end and the out-of-order execution back end, which is constituted by the remaining five stages. Dispatched instructions are stored in the reorder buffer and in the corresponding issue queue: the integer queue, the floating-point queue, or the load/store queue. The issue stage is responsible for waking up the instructions in the issue queues when they are ready and then selecting them for execution. Issued instructions read their operands from the register file (seventh stage), execute (eighth stage), and write back their results (ninth stage). The last stage commits the executed instructions in program order and updates the architectural state of the processor.

We have configured our simulator to model an aggressive eight-instruction-wide superscalar processor. The main values of this setup are shown in Table 1. We simulate three different state-of-the-art fetch architectures. Besides our stream fetch engine, we model the fetch target buffer (FTB) architecture [12] and the trace predictor architecture [9]. We model two versions of the latter: one using a trace cache and another one not using it. All of these architectures use an eight-entry FTQ [12] 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 has 64 Kbytes and uses wide cache lines, that is, four times the processor fetch width [1]. The trace predictor architecture using a trace cache is actually evaluated using a 32 Kbyte instruction cache, whereas the remaining 32 Kbytes are devoted to the trace cache. This hardware budget is equally divided into a filter trace cache [29] and a main trace cache. In addition, we use selective trace storage [30] to avoid trace redundancy between the trace cache and the instruction cache.

3.1 Benchmark Simulation

Our results are average measures collected from the simulation of 10 SPECint2000 benchmarks, which are listed in Table 2. 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. In addition, we excluded *252.eon* because we were unable to correctly optimize it.

The benchmarks were compiled on a DEC Alpha AXP 21264 processor with Digital Unix v4.0 using the standard DEC C v5.9-011 and Compaq C++ v6.2-024 compilers. The

optimization level is set to O2 and loop unrolling is deactivated. Since previous work [1], [2] 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 optimized code layout was generated using the Spike tool [13] 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.

In order to analyze the impact of inlining heuristics on the length of instruction streams, we need to modify the source code of the optimization tool. However, the source code of our Spike version is not available for doing such modifications. Therefore, we used the ALTO optimizer [21] to generate optimized binaries with different levels of inlining. ALTO still uses the same profile information collected with the Pixie v5.2 tool and the *train* input set.

Regardless of the case, all benchmarks are always simulated until completion. In order to explore a wide range of setups and binaries, we chose the inputs for the benchmark programs from the *MinneSPEC* [31] 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 to assure evaluation correctness. For those benchmarks that did not have an adequate *MinneSPEC* input available, we selected an input from the official *test* input set. The benchmark input set used is shown in Table 2.

3.2 Fetch Models

The stream fetch engine model [1], [2] is shown in Fig. 2a. The current fetch address is used to index the next-stream predictor and generate a prediction. Every stream prediction contains the stream length and the stream target. Both the fetch address and the stream length form a fetch request that is stored in the FTQ. This request will be used later to drive the instruction cache, obtain a line from it, and select which instructions from the line should be executed. Regarding the stream target, it will be the fetch address for the next prediction. Thus, fetch address generation makes it possible for the stream predictor to follow any speculative path without external assistance. A more detailed explanation is provided in Section 7.

For comparison purposes, we simulate two other state-of-the-art fetch architectures: the FTB and the trace cache. Our FTB model is similar to the one described in [12]: an FTB feeding the FTQ with prediction requests. Fig. 2b shows a diagram representing this fetch architecture. The main difference between our model and the original one is that we use a perceptron predictor [32] to provide accurate conditional branch predictions.

Our trace cache fetch model is similar to the one described in [10] but uses an FTQ [12] to decouple the trace predictor from the trace cache, as shown in Fig. 2c. Trace predictions are stored in the FTQ, which feeds the trace cache with trace identifiers. An interleaved branch target buffer (BTB) is used to build traces in 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 two-way interleaved instruction cache is used to allow traces to be built as fast as possible. This mechanism

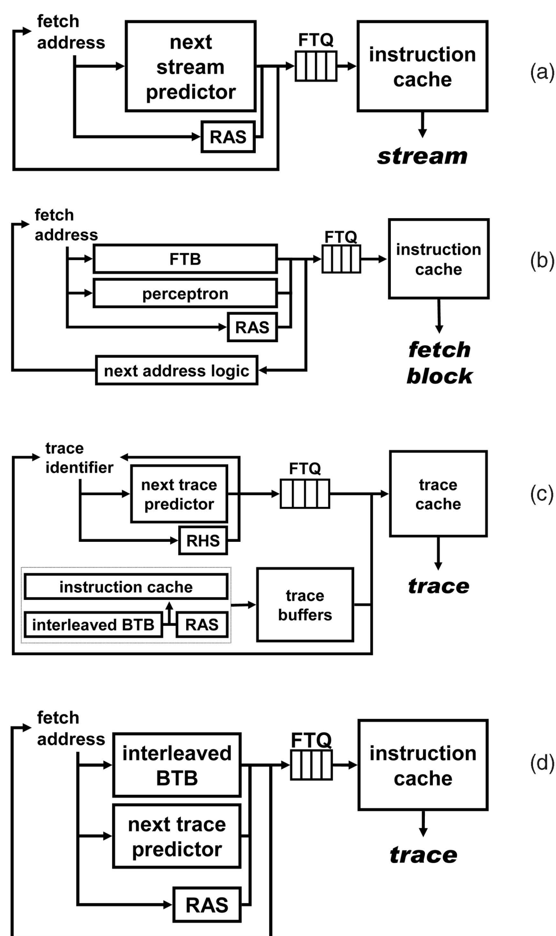

Fig. 2. Fetch models evaluated. (a) Stream fetch engine. (b) FTB fetch engine. (c) Trace fetch engine (with a trace cache). (d) Trace fetch engine (without a trace cache).

is able to obtain up to a full cache line in a cycle, independent of PC alignment.

Nevertheless, as shown in [10], it is not strictly necessary to implement the trace predictor coupled with a trace cache. Fig. 2d shows a fetch model that uses a trace predictor coupled with a conventional instruction cache. The trace predictor provides multiple branch direction predictions, that is, it provides the fetch PC of the first basic block in the predicted trace followed by several 1-bit taken/not-taken predictions. An interleaved BTB is used to provide the information about the basic blocks beyond the first one that are included in every multiple prediction. These data are stored in the FTQ and then used to access the instruction cache and obtain the corresponding instructions.

All of these fetch architectures use specialized structures to predict return instructions. The FTB and the stream fetch architecture use a return address stack (RAS) [33] to predict the target address of return instructions. There are actually two RASs: one updated speculatively in the prediction stage and another one updated nonspeculatively in the commit stage. The latter is used to restore the correct RAS state in case of a branch misprediction. The FTB fetch architecture also uses a cascaded structure [34] to improve the prediction accuracy of the rest of the indirect branches. Both the stream predictor and the trace predictor are able to correctly predict them because they use a similar cascaded structure.

TABLE 3
Configuration of the Simulated Branch Predictors

FTB fetch architecture (approx. 50KB)		
perceptron predictor	FTB	1-cycle predictor
256 perceptrons 4096x14 bit local and 40 bit global history	L1: 1024-entry, 4-way L2: 4096-entry, 4-way DOLC 14-2-4-10	64-entry gshare 6-bit history 32-entry BTB
Stream fetch architecture (approx. 32KB)		
next stream predictor		1-cycle predictor
L1: 1024-entry, 4-way L2: 4096-entry, 4-way DOLC 16-2-4-10		32-entry spred DOLC 0-0-0-5
Trace fetch architecture (approx. 80KB)		
next trace predictor	interleaved BTB	1-cycle predictor
L1: 2048-entry, 4-way L2: 4096-entry, 4-way DOLC 10-4-7-9	L1: 1024-entry, 4-way L2: 4096-entry, 4-way DOLC 14-2-4-10	32-entry tpred DOLC 0-0-0-5 perfect BTB

The trace fetch engine uses a return history stack (RHS) [9] instead of an RAS. This mechanism is more efficient than an 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 RHSs: one updated speculatively in the prediction stage and another one updated nonspeculatively in the commit stage. The latter is used to restore the correct RHS state in case of a branch misprediction. However, the RHS in the trace fetch architecture is less accurate in predicting return instructions than the RAS in the rest of the 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 simulated branch predictors, varying their size from small and fast tables to big and slow tables. We use realistic prediction table access latencies calculated using the CACTI 3.0 tool [35]. We modified CACTI to model tagless branch predictors and to work with setups expressed in bits instead of bytes. The data we have obtained corresponds to a 0.10μm technology. For translating the access time from nanoseconds to cycles, we assume an aggressive 8-fan-out-of-four-delay clock period, that is, a 3.47 GHz clock frequency, as reported in [3]. It has been claimed in [36] that eight fan-out-of-four delays is the optimal clock period for integer benchmarks in a high-performance processor implemented using a 0.10μm technology.

We have found that the best performance is achieved by using three-cycle latency tables [6]. 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 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 configuration of the simulated predictors. We have explored a wide range of history lengths, as well as predictor index setups,[1] and selected the best one found for each configuration. Table 3 also shows the approximate 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 fewer hardware resources because it uses a single prediction mechanism, whereas the other evaluated fetch architectures use some separate structures.

Our fetch models also use an overriding mechanism [4], [5] to complete a branch prediction in each cycle. A small branch predictor, supposed to be implemented using very fast hardware, generates the next fetch address in a single cycle. Although fast, this predictor has low accuracy, so the main predictor is used to provide an accurate backup prediction. This prediction is obtained three cycles later and compared with the prediction provided by the single-cycle predictor. If the 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 Including SFBs in Instruction Streams

On the average, 55 percent of dynamic streams are finished by taken conditional branches. Enlarging streams beyond these branches would allow improving fetch bandwidth, increasing branch prediction accuracy, and reducing branch predictor energy consumption. In this section, we describe a technique that includes forward conditional branches in instruction streams.

Including a forward conditional branch into a stream can be understood as a type of hardware predication [17]. Our technique nullifies the instructions skipped by the branch, preventing them from modifying the processor state. If the forward conditional branch is mispredicted, all of the speculative work done after it is discarded. However, the skipped instructions remain in the processor and, thus, it is not necessary to fetch them again, saving fetch bandwidth. There are several proposals in the literature that are close to ours, like the Collapsing Buffer [37], Skipper [38], and some optimizations proposed for trace cache architectures [39]. Wish Branches [40] is a more recent proposal that uses predication for hard to predict branches and branch prediction for easy to predict ones. The technique presented in this section is similar in spirit to these works but uses a next-stream predictor to guide the decision about when predication should be applied.

4.1 SFBs

Our proposal is aimed at reducing the impact of a possible misprediction without requiring an increase in the fetch engine complexity. The idea is to select SFBs, that is, those conditional branch instructions whose target is a few instructions ahead in the code layout. When a taken SFB is found, the instruction stream is not finished by it. Instead, 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

1. We use the DOLC scheme presented in [9] to index the cascaded FTB, the stream predictor, and the trace predictor.

2. The first level of the trace and stream predictors, as well as the first level of the cascaded FTB, is actually smaller than the second one because larger first-level tables do not provide significant improvements in prediction accuracy.

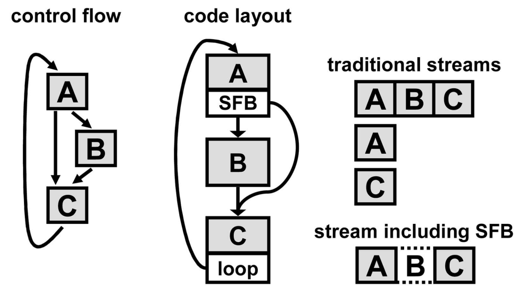

Fig. 3. Example of an instruction stream containing an SFB.

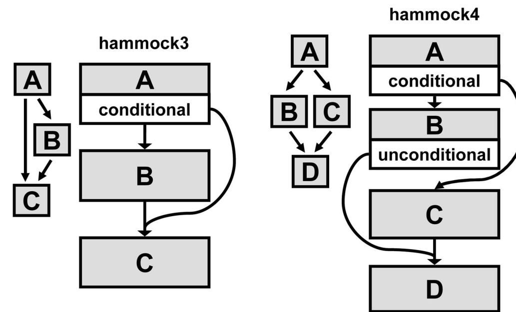

Fig. 4. A *hammock3* structure (if-then structure without an else clause) and a *hammock4* structure (if-then-else structure).

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, that is, 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 the instructions again after the SFB target.

Fig. 3 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 (for example, using profiling) that it has a high probability of being mispredicted. In this case, basic blocks ABC form a new stream that 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 that 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 ignored, that is, it 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.

4.2 Viability Evaluation

The potential of our SFB technique depends on the amount of branches that can take advantage of it. A branch can benefit from our technique if it is a forward conditional branch with no other branch instructions inside the static code from the branch itself to its target, that is, the portion of the code skipped when the branch is taken. Such a branch corresponds to an if-then high-level structure. Fig. 4 shows an example of this 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 structure like the one shown in Fig. 4 can also exploit 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 structure can take advantage of our SFB mechanism by 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 in nonoptimized codes to find *hammock3* and *hammock4* structures [41]. We have omitted results using optimized codes because code layout optimizations tend to remove these structures. The optimization process could be done by taking into account the needs of our SFB mechanism, but this is left for future work. Overall, our results show that 2.5 percent of dynamic branches correspond to *hammock3* structures and 2.7 percent of dynamic branches correspond to *hammock4* structures. This means that only 5.2 percent of dynamic branches can benefit from our SFB mechanism. We have also found that 6.4 percent of dynamic mispredictions are caused by *hammock3* branches and 6.5 percent of mispredictions are caused by *hammock4* branches. Therefore, this mechanism can only be used to reduce the impact of 12.9 percent of dynamic branch mispredictions and, thus, it has little potential to improve processor performance.

5 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 at a loop branch whose target is the starting address of the stream itself.

Many mechanisms have been developed in the literature to predict the behavior of loops. Joseph and Vajapeyam [42] propose control-flow prediction to detect loop structures and use this information to improve branch prediction accuracy. Sherwood and Calder [43] use a Loop Termination Buffer (LTB) to predict patterns of loop branches

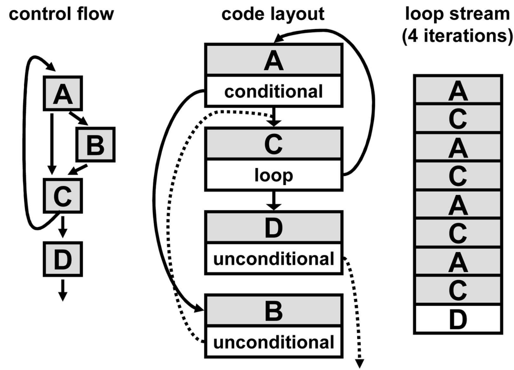

Fig. 5. Example of a loop stream.

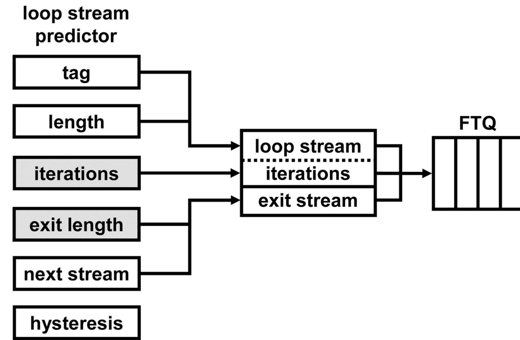

Fig. 6. The loop-stream predictor.

(backward branches). The LTB stores the number of times that a loop branch is taken and a loop iteration counter is used to store the current iteration of the loop. De Alba and Kaeli [44] use a mechanism to predict several loop characteristics: the internal control flow, the number of loop visits, the number of iterations per loop visit, the dynamic loop body size, and the patterns leading up to the loop visit. Nevertheless, to the best of our knowledge, our proposal is the first one that uses loop prediction for tolerating the prediction table access latency instead of for improving prediction accuracy.

5.1 The Loop-Stream Predictor

Fig. 5 shows an example of a loop stream and its code layout. The structure shown is a loop containing an if-then structure. 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 A and C. 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 the fetch address, the original next-stream predictor [1], [2] is able to predict the stream AC and basic block A as its target address. If this predictor is enhanced with the ability to predict 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 A and C 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 by 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 the fetch address, the loop-stream predictor is able to predict stream $ACACACACD$, which is composed by nine basic blocks.

In order to achieve it, we should add new fields to the stream prediction tables, as shown in Fig. 6. Given a fetch address, the original stream prediction table [1], [2] is looked up to find an entry whose tag matches with the fetch address. If an entry with the appropriate tag is found, it contains a length field that indicates the number of instructions that compose the stream starting at that address. The entry also contains a target address field, which will be used as the fetch address in the next cycle, and a hysteresis counter that is used to decide whether or not a stream is replaced from the prediction table. A more in-depth description of the stream prediction tables can be found in Section 7.

In case of a loop stream, 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 address of the stream that appears after the last iteration of the loop, 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 [35], 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, as shown in Fig. 6. When a loop stream is predicted, the FTQ is fed by the loop-stream prediction registers. The tag and length fields are used to form the loop stream, which has its own starting address 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 used to generate the stream that should be executed after the loop stream, that is, when all iterations have been fetched. The first instruction of this exit stream is the one following the last instruction of the loop stream.

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 of the data in the loop-stream prediction registers have been used to feed the FTQ, the stream predictor starts generating new predictions again until the next loop-stream prediction is found.

5.2 Loop-Stream Prediction Evaluation

We have done an in-depth evaluation of the loop-stream predictor using both the baseline codes and the codes optimized using Spike [13]. As can be expected, there is a larger percentage of loop streams in the optimized codes since it is less likely that a taken branch appears inside a loop

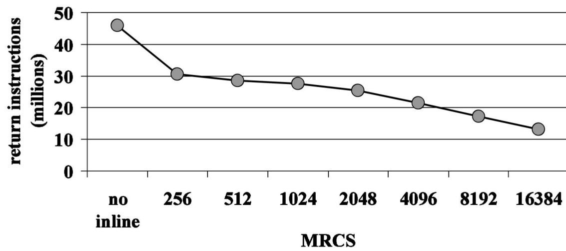

Fig. 7. Average number of return instructions (millions), varying MRCS from 256 to 16,384 instructions.

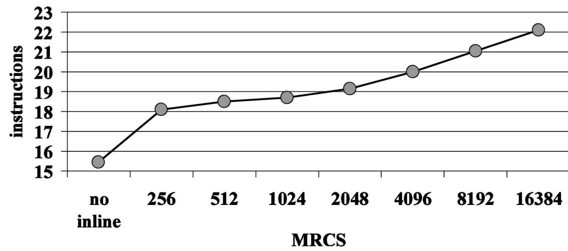

Fig. 8. Average stream length, varying MRCS from 256 to 16,384 instructions.

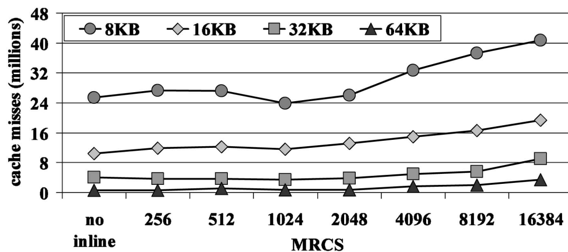

Fig. 9. Average instruction cache misses (millions), varying MRCS from 256 to 16,384 instructions.

body, breaking the loop stream. Despite this, the average percentage of loop streams is not too high: 7 percent using the baseline code and 14 percent using the optimized code.

However, the main problem of our loop-stream predictor is not the low portion of code covered by loop streams. Our results show that, on average, loop streams contain around 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 possible a reduction in the branch predictor energy consumption but will not improve the ability to tolerate 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-stream predictor setups and we have found that the best one achieves a prediction accuracy that is very similar to the original stream predictor. However, we have measured that, when loop-stream prediction is enabled, a processor model suffering from a realistic prediction delay achieves less than 1 percent of performance speedup in most cases. Moreover, we have found, using CACTI [35], that the reduction in the stream predictor activity does not compensate for the increase in the prediction table size caused by the new fields required. The loop-stream predictor consumes 0.5 percent more energy than the original stream predictor when using optimized codes and 8 percent more energy when using baseline codes. These results suggest that, in order to enlarge instructions streams, different research lines should be examined.

6 Aggressive Procedure Inlining

The ALTO [21] optimizer is able to perform aggressive procedure inlining. This 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.

The higher the MRCS value is, the more aggressive is the procedure inlining performed. As a measure of how effective procedure inlining is, Fig. 7 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 16,384 instructions, and compare it against the baseline code. 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 of over 70 percent of return instructions against the code optimized without inlining.

6.1 The Impact of Inlining on the Length of Instruction Streams

The aggressiveness of procedure inlining has a direct impact on the length of instruction streams, as shown in Fig. 8. By definition, instruction streams are limited by taken branches. Procedure inlining removes a large amount of function calls and return instructions, that is, it removes a large amount of taken branches, which involves an enlargement of instruction streams. Although this enlargement is limited by the removal of instructions associated with the procedure call overhead, the overall effect is that the more aggressive the inlining is, the longer the instruction streams are.

However, this enlargement is not free. Aggressive inlining duplicates large portions of the program code, increasing the number of instruction cache misses. Fig. 9 shows the average number of instruction cache misses, varying the MRCS value from 256 to 16,384 instructions and the total instruction cache hardware budget from 8 to 64 Kbytes. Although, in general, more aggressive inlining involves a higher number of instruction cache misses, it is not always a direct relationship. For example, the 8 Kbyte instruction cache suffers from fewer misses using a 1,024-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

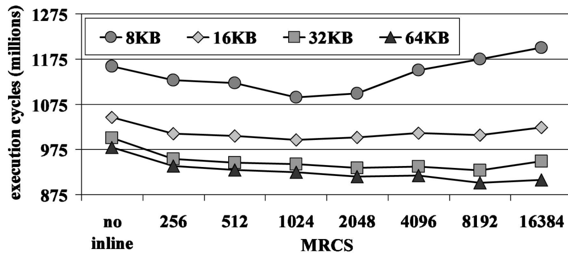

Fig. 10. Average performance of an 8-wide processor, varying MRCS from 256 to 16,384 instructions.

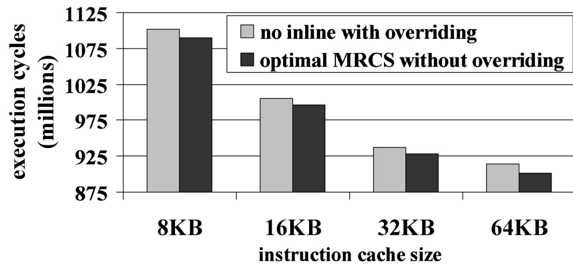

Fig. 11. Average performance of an 8-wide processor with prediction overriding but not using inlining and without prediction overriding but using inlining and the optimal MRCS value.

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 performed by ALTO, like dead code elimination or code scheduling, and potentially reducing the number of instruction cache misses. Nevertheless, it only happens for intermediate values of the MRCS parameter, that is, when the impact of the larger code size 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 stream predictor ability to tolerate the prediction table access latency, and thus improving the processor performance. On the other hand, aggressive inlining increases the number of instruction cache misses, which degrades the processor performance. In order to find the optimal setup, we explore this trade-off in the next section.

6.2 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.

Fig. 10 shows the processor performance using a realistic prediction table access latency (three cycles) without prediction overriding. We vary the MRCS value from 256 to 16,384 instructions and the total instruction cache hardware budget from 8 to 64 Kbytes. The bigger the cache is, the more aggressive the inlining that can be performed is. Thus, the optimal value of MRCS is higher for the bigger cache sizes. Both the 8 Kbyte and 16 Kbyte instruction caches achieve their optimal performance using a 1,024-instruction MRCS value, whereas the 32 Kbyte and 64 Kbyte instruction caches achieve their optimal performance using a more aggressive 8,192-instruction MRCS value.

The best performance is achieved by the 64 Kbyte instruction cache due to its lower miss rate. Using this cache setup, the code inlined using the optimal MRCS value achieves 8 percent performance improvement over the noninlined code. However, this improvement is not necessarily caused by the ability to tolerate 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 percent performance improvement over the noninlined code. Since the predictor delay is not a problem for the ideal latency predictor, this improvement cannot be attributed to the ability to tolerate 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. Consequently, the additional 2 percent performance improvement is caused by the ability of longer streams to hide the prediction table access latency.

Fig. 11 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 a code optimized without inlining, even if it uses prediction overriding.

However, using a 64 Kbyte instruction cache, the processor without overriding executing an inlined code achieves just 1.5 percent reduction in the total number of execution cycles over the processor executing a noninlined 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 an inlined code would be unable to outperform the processor with overriding executing noninlined code if the 2 percent improvement provided by hiding the predictor latency is not taken into account.

7 Multiple-Stream Prediction

Previous techniques enlarge instruction streams, but they are not able to achieve good performance results. These numbers show that focusing on particular stream types is not a correct approach to enlarge instruction streams due to Amdahl's law: Although these techniques enlarge a set of instruction 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 achieving this is the multiple-stream predictor.

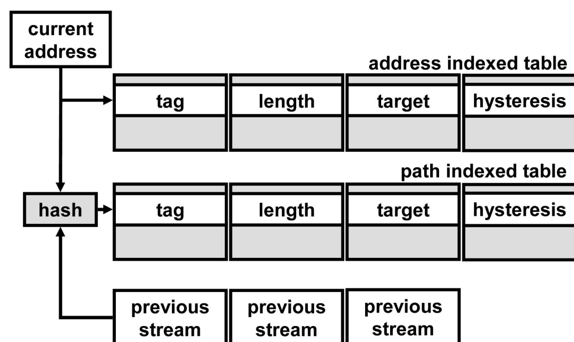

Fig. 12. The next-stream predictor design.

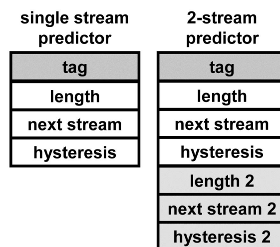

Fig. 13. Fields required by the original single-stream predictor and by a multiple-stream predictor that is able to provide two streams per prediction.

7.1 The Stream-Predictor Design

The next stream predictor [1], [2] is organized like a set associative cache using an LRU replacement policy. Each predictor entry contains a tag, the stream length, the stream target, and a hysteresis counter. The fetch address is used to index the prediction table and the tag is checked to ensure a predictor hit. The stream length is used to determine which instructions should be executed from that point onward. Finally, the stream target is the next fetch address, that is, it will be used as the fetch address in the next cycle. If a stream starting at the fetch address is not found in the prediction table, sequential instructions will be fetched until a predictor hit or a fetch redirection after a misprediction.

To accurately predict streams, our predictor uses correlation with previously executed streams. The fetch address and the contents of a history register with the starting addresses of previous streams are hashed together to obtain an index into the prediction table. The use of path correlation makes it possible for the stream predictor to store several different streams starting at the same address, that is, overlapping streams, as well as multiple target addresses for each of them.

In general, a longer history involves an improvement in branch prediction accuracy. However, long histories also involve more entries used by each stream and, thus, more aliasing in the prediction table, preventing other streams from staying in the table. In addition, long histories increase the prediction table learning time. Although these effects limit the gain achieved by long histories, profile information has revealed that not all streams need path correlation to be accurately predicted [2].

Fig. 12 shows a cascaded implementation of the next-stream predictor that takes advantage of this fact. The prediction table is divided in two: a first-level table indexed only by the fetch address and a second-level table indexed using path correlation. Those streams that do not need correlation are kept in the first-level table, preventing them from using correlation and therefore avoiding unnecessary aliasing. The remaining streams are kept in the second-level table, using path correlation. This design not only avoids unnecessary aliasing but also allows partially hiding the learning times caused by long histories because, while the second-level table is being trained, the first-level one is already able to generate stream predictions.

The cascaded stream predictor works as follows: The two tables make their predictions in parallel. The second-level table is supposed to be more accurate than the first-level table due to the use of path correlation. Therefore, if the second-level table has information to make a prediction, it will be used. Otherwise, the prediction of the first-level table is used.

A stream is introduced in both tables the first time it appears. The next time its starting address appears, the first-level will provide a prediction, but, if the history is different, the second-level will not be able to do it. If the first-level prediction is correct, the stream is updated only in the first-level table. Otherwise, it is introduced again in the second-level table. Therefore, if the first-level table is not able to provide accurate predictions, the second-level table will be warmed up until it contains enough information to generate predictions. On the contrary, the streams that do not need correlation to be accurately predicted are introduced only once in the second-level table, avoiding unnecessary aliasing. The second-level entry assigned to a stream that does not need correlation would be dynamically replaced from the table by other streams without degrading the prediction accuracy since the first-level will still be able to correctly predict it.

The hysteresis counters are used to decide whether a stream should be replaced from one of the prediction tables. When a prediction table 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. The counters saturate at their maximum value. However, if the new stream does not match with the stream already stored in the entry, the counter is decreased. 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.

7.2 The Multiple-Stream Predictor

The objective of our multiple-stream predictor is to predict 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, making it possible to tolerate the prediction table access latency.

Fig. 13 shows the additional fields required by a two-stream predictor. A single tag field is enough to ensure prediction table hits. The tag value corresponds to the starting address of the stream sequence. The rest of the fields should be duplicated: the stream length, the stream target, and the hysteresis counter. The tag and the first

length field determine the first stream that should be executed. The target of this stream, determined by the first target field, is the starting address of the second stream, whose length is given by the second length field. The second target field is the target of the second stream and, thus, the actual next fetch address.

The main drawback of this technique is that providing multiple streams per prediction involves increasing 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.

Since it is important to take maximum advantage of the available space in the first-level table, we use the hysteresis counters to detect frequently executed stream sequences. Every stream in a sequence has a hysteresis counter associated to it which behaves like the counter used by the original stream predictor. We have found that 3-bit hysteresis counters, increased by one and decreased by two, provide the best results.

When the prediction table is looked up, the first stream is always provided. The second stream is only provided if the corresponding hysteresis counter is saturated. 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 both hysteresis counters have saturated, then a frequently executed sequence has been identified and it will be provided by the multiple-stream predictor.

In this way, a single prediction table lookup may provide two separate stream predictions which are supposed to be executed sequentially. After a multiple-stream prediction, both streams in the predicted sequence are stored separately in the FTQ, which involves using the multiple-stream predictor not requiring additional changes in the processor front end. Extending this mechanism for predicting three, four, or more streams per sequence is straightforward, but we have found that predicting sequences longer than two streams does not provide a significant performance improvement.

7.3 Multiple-Stream Prediction Evaluation

In this section, we evaluate the multiple-stream predictor and compare it with our baseline fetch models: the FTB fetch architecture [12], the original stream predictor [1], [2], and the trace predictor [9] both isolated and coupled with a trace cache.

First of all, we analyze the amount of instructions provided per prediction. Then, we evaluate the accuracy of these predictions. We show later how these two factors combine to hide the predictor access latency and thus to improve processor performance. Finally, we analyze the complexity of the fetch architectures evaluated both in terms of chip area and energy consumption.

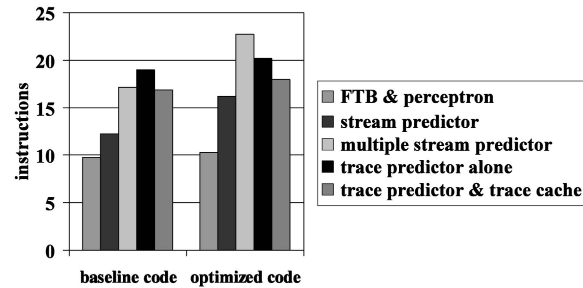

Fig. 14. Average number of instructions per prediction provided by the evaluated fetch architectures.

7.3.1 *Instructions per Prediction*

Increasing the total number of instructions per prediction makes it possible to hide the branch predictor access latency, improving processor performance. Fig. 14 shows the average amount of instructions per prediction provided by all our fetch models. Data is presented for both the baseline code layout and the code layout optimized with Spike [13].

Using the baseline code, the higher amount of instructions per prediction is provided by the trace predictor. Traces are formed by any basic block, regardless of whether the branch terminating it is taken or not. Thus, they are longer than streams or FTB fetch blocks, which contain just one taken branch.

The trace predictor alone provides a higher amount of instructions per prediction because the hardware implementation of the trace cache limits the maximum trace size. However, the trace predictor alone provides just 12 percent more instructions per prediction than the trace predictor coupled with a trace cache because there are additional trace size limitations. In order to provide accurate predictions, traces must be finalized when one of these branch types is found: a loop branch, a function call or return, and any indirect branch [9], [45]. Not applying these heuristics would involve poor prediction accuracy and, thus, severe performance losses.

Our multiple-stream predictor is also able to provide a high amount of instructions per prediction, which is comparable to the trace cache. In spite of being limited to two taken branches, our multiple streams are just 11 percent shorter that the traces provided by the trace predictor alone. These results show that although traces may have up to 10 taken branches in our model, the average trace contains just two to three taken branches due to the trace formation heuristics.

Code layout optimizations allow increasing the length of instruction streams. These 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. The impact is lower on FTB fetch blocks and instruction traces [2]. On one hand, FTB fetch blocks are finalized by branches that have been taken at least once, so reducing the number of times these branches are taken is less relevant. On the other hand, instruction traces ignore taken branches and, thus, they take little advantage from these optimizations.

On the average, FTB fetch blocks and instruction traces are just 6 percent longer when using code layout optimizations, whereas streams are 30 percent longer when using them. Using optimized codes, the multiple-stream predictor

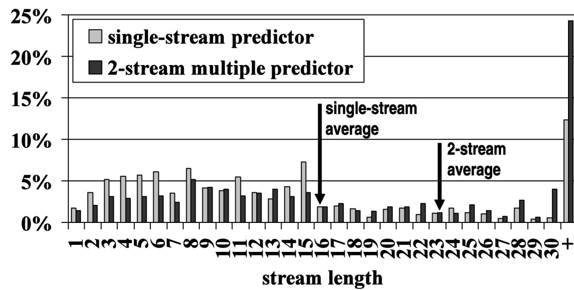

Fig. 15. Distribution of dynamic predictions according to the amount of instructions provided when using a single-stream predictor and a two-stream predictor.

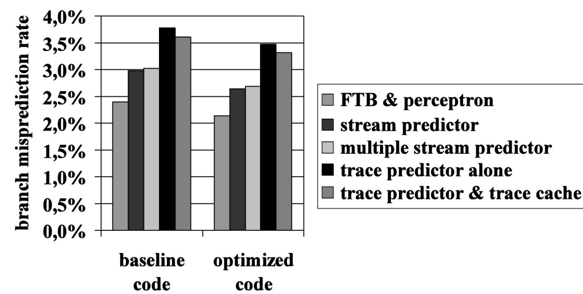

Fig. 16. Branch misprediction rate for the evaluated fetch architectures.

provides the highest number of instructions per prediction: 13 percent more instructions than the trace predictor alone and 40 percent more instructions than the original stream predictor.

However, having long average predictions does not involve most multiple streams being long. Some streams could be long, providing high fetch bandwidth, whereas other streams could be short, degrading the potential performance. Therefore, the distribution of dynamic stream lengths should be analyzed.

Fig. 15 shows a histogram of the amount of instructions provided per prediction using the optimized codes. 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. The data presented are average results collected from our 10 benchmarks optimized using Spike [13]. These data are shown for both the original single-stream predictor and the two-stream predictor.

Using the original stream predictor, most streams are shorter than the average length: 70 percent of dynamic streams have 16 or less instructions. These short streams are the main limitation to performance. For example, 30 percent of dynamic streams have less than eight instructions. If we consider our 8-wide execution core, streams shorter than eight instructions are wasting resources since they do not provide enough instructions to take advantage of the whole fetch bandwidth.

The 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. Furthermore, there is an impressive increase in the percentage of predictions that provide more than 30 instructions. From these results, we conclude that the multiple-stream predictor is a good strategy to provide enough instructions to hide the prediction delay, especially when using optimized codes.

7.3.2 Prediction Accuracy

Providing a high amount of instructions per prediction would be useless if they do not belong to the correct execution path. Therefore, it is also crucial to provide accurate branch prediction. Fig. 16 shows the average branch misprediction rate for the five evaluated fetch architectures and for both code layouts.

Although the FTB fetch architecture is the one that provides fewer instructions per prediction, it is the most accurate due to the perceptron algorithm. Applying this algorithm to the stream or trace predictors would improve their prediction accuracy, but this is out of the scope of this paper. In addition, the perceptron implementation is more complex since it requires a hardware adder-tree structure.

The stream predictor is a simpler design because it relies just on prediction tables, not requiring a complex indexing function. Although the perceptron has lower misprediction rate, the stream predictor still provides good prediction accuracy, suffering from less than 3 percent misprediction rate. In addition, code layout optimizations improve the stream prediction accuracy. Instruction streams are longer in optimized codes, so most of the program execution is held in a lower number of streams and, thus, aliasing is reduced in the prediction tables.

Our multiple-stream predictor does not sacrifice accuracy to provide more instructions per prediction since it is as accurate as the single-stream predictor. Moreover, both the single and the multiple-stream predictors are more accurate than the trace predictor: They have a 15 percent lower misprediction rate than the trace cache using a trace predictor and a 20 percent lower misprediction rate than the trace predictor alone.

Trace prediction is less accurate than stream prediction because trace limits are selected at arbitrary points in program execution, whereas streams adapt to high-level structures such as hammocks and loops. Consequently, the multiple-stream predictor achieves a good balance between prediction accuracy and instructions provided per prediction.

7.3.3 Processor Performance

The branch prediction accuracy and the amount of instructions provided per prediction combine to determine the ability to hide the branch predictor access latency, thus increasing performance. Fig. 17 shows the average processor performance achieved by the five evaluated fetch architectures for both the baseline and the optimized code layout. Besides the performance of the fetch engines using overriding (stripped bars), Fig. 17 shows the performance achieved when overriding is not used (full bars). Performance is measured in execution cycles, so the lower, the better.

The trace predictor provides better performance than the FTB fetch architecture and the original stream fetch engine [1], [2]. The trace predictor achieves slightly better performance when it is coupled with a trace cache since it allows fetching instructions beyond a taken branch at a faster rate, although this is done at the cost of increasing complexity. Nevertheless, it is interesting to note that the original stream fetch engine achieves almost the same performance as the trace cache when using optimized codes since streams are longer when using these codes.

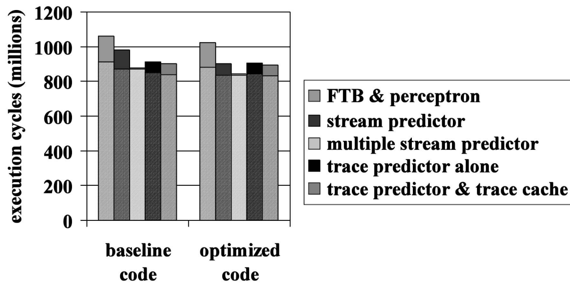

Fig. 17. Processor performance when using (stripped bars) and not using (full bars) overriding.

Our multiple-stream predictor is also able to provide good performance results. Using the baseline codes and prediction overriding, the multiple-stream predictor achieves a similar performance to the original stream predictor, which is close to the trace predictor performance. Furthermore, the most important observation is that the multiple-stream predictor provides the best performance when overriding is not used. Optimized codes present a similar behavior: The multiple-stream predictor provides performance close to the trace predictor when overriding is used, whereas it is the best one when prediction overriding is not used.

These great performance results are achieved because our multiple-stream predictor is able to hide the prediction delay as efficiently as overriding does. It can be seen in Fig. 17 that the multiple-stream predictor achieves almost the same performance regardless of whether overriding is used or not, whereas all of the remainder fetch architectures evaluated lose performance when overriding is not used.

7.3.4 Chip Area and Energy Consumption

We have used our modified CACTI tool [35], configured with 0.10μm technology parameters, to model all of the structures required by the five fetch architectures evaluated in this section. This tool allows estimating the area and energy consumption of all fetch structures, including the prediction tables, the instruction and trace caches, the FTQ, and so forth. Although our tool is not able to model interconnection wires between these structures, we consider that they would require less area and energy for the stream fetch engine due to its lower design complexity.

Trace predictor complexity comes mainly from the need for a second fetch engine to build traces. When there is a miss in the trace cache, instructions must be fetched, decoded, and packed into a trace from a secondary source. This second fetch engine increases cost, complexity, and area compared to a system that always fetches instructions from the same location, like the FTB or the stream predictor. The trace predictor alone is less complex than the trace predictor coupled with the trace cache because it does not need separate instruction and trace caches, but it still needs the secondary fetch engine to provide basic-block data according to the trace predictions.

Overall, our estimations indicate that the stream fetch engine is the simplest fetch architecture evaluated. The multiple-stream predictor requires higher complexity since the size of the first-level table is increased. Fortunately, the tag array is unmodified and no additional access port is required. Therefore, the multiple-stream predictor requires just 8 percent more area than the original one.

The multiple-stream predictor also consumes more energy than the original one since the first-level prediction table is bigger. However, this is compensated for by the reduction in the total number of prediction table accesses due to the fact that each prediction provides more instructions. The ability to provide two streams per prediction causes a 28 percent reduction in the total number of prediction table lookups and updates, reducing the additional energy consumption to just 15 percent.

Despite this, the multiple-stream predictor requires less area and consumes less energy than the other fetch architectures evaluated since its design is simpler. The multiple-stream predictor requires 7 percent less chip area than the FTB fetch architecture, which uses a separate conditional branch predictor. The multiple-stream predictor also requires 20 percent less area than the trace predictor alone and 29 percent less area than the trace predictor coupled with the trace cache. Furthermore, the multiple-stream predictor consumes 15 percent less energy than the trace predictor alone, 19 percent less energy than the FTB fetch architecture, and even 42 percent less energy than the trace predictor coupled with the trace cache. These results highlight that the multiple-stream predictor presents a better balance between performance and complexity.

8 Conclusions

Current technology trends create new challenges for the fetch architecture designers. Higher clock frequencies and larger wire delays cause branch prediction tables to require multiple cycles to be accessed [3], [4], limiting the fetch engine performance. This fact has led to the development of complex hardware mechanisms like prediction overriding [4], [5] to hide the prediction table access delay.

In order to hide the prediction table access delay without increasing the fetch engine complexity, we propose using long instruction streams [1], [2] as the basic prediction and fetch unit. 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. However, our first attempts to enlarge instruction streams do not provide good results. We present a technique for including SFBs in instruction streams, but it is applicable to a small percentage of existing branches and, thus, it would have negligible impact on performance. We also present a loop-stream predictor which enlarges streams by combining loop iterations, but it tends to enlarge streams that are already long enough to tolerate the prediction table access latency. Finally, we show that aggressive procedure inlining removes a high percentage of function calls and returns, enlarging streams. However, the 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.

Consequently, focusing on particular branch types is not a correct approach for enlarging instruction streams. The aforementioned techniques enlarge some streams, but the remaining not-enlarged streams limit the achievable benefit

according to Amdahl's law. In order to overcome this problem, we propose combining frequently executed streams into long virtual streams using a multiple-stream predictor. Streams are combined regardless the type of the branch finalizing them, capturing the behavior of any frequently executed code structure. The captured structures may include SFBs, loop streams, and procedure calls. Nevertheless, the multiple-stream predictor is not limited to them and, thus, it is a more general approach.

Our novel predictor design provides predictions that contain, on the average, more than 20 instructions. The high amount of instructions per prediction provided by the multiple-stream predictor combines with its high prediction accuracy to allow our proposal to achieve performance results that are comparable to state-of-the-art fetch architectures. In addition, the multiple-stream predictor does not need a hardware overriding mechanism to hide the branch prediction table access latency. As a consequence, our design requires less chip area and consumes less energy than the previous proposals evaluated in this paper. We conclude from this that the multiple-stream predictor is a worthwhile alternative for the design of latency-tolerant fetch architectures due to its high performance and its relatively low complexity.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under Contract TIN-2004-07739-C02-01, the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC), the Barcelona Supercomputing Center, and an Intel fellowship. The authors would like to thank Ayose Falcón for his contributions to this research, as well as Manel Fernández and David Kaeli for their worthwhile help with the ALTO optimization tool.

References

[1] A. Ramirez, O.J. Santana, J.L. Larriba-Pey, and M. Valero, "Fetching Instruction Streams," *Proc. 35th Int'l Symp. Microarchitecture*, 2002.

[2] O.J. Santana, A. Ramirez, J.L. Larriba-Pey, and M. Valero, "A Low-Complexity Fetch Architecture for High-Performance Superscalar Processors," *ACM Trans. Architecture and Code Optimization*, vol. 1, no. 2, 2004.

[3] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00)*, 2000.

[4] D.A. Jimenez, S.W. Keckler, and C. Lin, "The Impact of Delay on the Design of Branch Predictors," *Proc. 33rd Int'l Symp. Microarchitecture*, 2000.

[5] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '02)*, 2002.

[6] O.J. Santana, A. Ramirez, and M. Valero, "Latency Tolerant Branch Predictors," *Proc. Int'l Workshop Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2003.

[7] O.J. Santana, A. Ramirez, and M. Valero, "Reducing Fetch Architecture Complexity Using Procedure Inlining," *Proc. Eighth Ann. Workshop Interaction between Compilers and Computer Architectures (INTERACT '04)*, 2004.

[8] O.J. Santana, A. Ramirez, and M. Valero, "Multiple Stream Prediction," *Proc. Sixth Int'l Symp. High Performance Computing (ISHPC '05)*, 2005.

[9] Q. Jacobson, E. Rotenberg, and J.E. Smith, "Path-Based Next Trace Prediction," *Proc. 30th Int'l Symp. Microarchitecture*, 1997.

[10] E. Rotenberg, S. Bennett, and J.E. Smith, "A Trace Cache Microarchitecture and Evaluation," *IEEE Trans. Computers*, vol. 48, no. 2, Feb. 1999.

[11] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line," US patent 5,381,533, 1995.

[12] G. Reinman, T. Austin, and B. Calder, "A Scalable Front-End Architecture for Fast Instruction Delivery," *Proc. 26th Int'l Symp. Computer Architecture (ISCA '99)*, 1999.

[13] R. Cohn, D. Goodwin, P.G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables," *Proc. Usenix Windows NT Workshop*, 1997.

[14] J.R. Ellis, *BULLDOG: A Compiler for VLIW Architectures*, ACM Doctoral Dissertation Awards. MIT Press, 1986.

[15] W.M.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing*, vol. 7, nos. 1-2, 1993.

[16] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann, "Effective Compiler Support for Predicated Execution," *Proc. 25th Int'l Symp. Microarchitecture*, 1992.

[17] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Proc. 10th Symp. Principles of Programming Languages (POPL '83)*, 1983.

[18] R. Allen and S. Johnson, "Compiling C for Vectorization, Parallelization, and Inline Expansion," *Proc. Conf. Programming Language Design and Implementation (PLDI '88)*, 1988.

[19] W.W. Hwu and P.P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. Conf. Programming Language Design and Implementation (PLDI '89)*, 1989.

[20] A. Ayers, R. Gottlieb, and R. Schooler, "Aggressive Inlining," *Proc. Conf. Programming Language Design and Implementation (PLDI '97)*, 1997.

[21] 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*, vol. 31, no. 1, 2001.

[22] H. Aydin and D. Kaeli, "Using Cache Line Coloring to Perform Aggressive Procedure Inlining," *ACM Computer Architecture News*, vol. 28, no. 1, 2000.

[23] D.A. Jimenez, "Reconsidering Complex Branch Predictors," *Proc. Ninth Int'l Conf. High-Performance Computer Architecture (HPCA '03)*, 2003.

[24] A. Seznec and A. Fraboulet, "Effective Ahead Pipelining of Instruction Block Address Generation," *Proc. 30th Int'l Symp. Computer Architecture (ISCA '03)*, 2003.

[25] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95)*, 1995.

[26] A. Falcón, O.J. Santana, A. Ramirez, and M. Valero, "Tolerating Branch Predictor Latency on SMT," *Proc. Fifth Int'l Symp. High Performance Computing (ISHPC '03)*, 2003.

[27] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report*, vol. 10, no. 14, 1996.

[28] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95)*, 1995.

[29] R. Rosner, A. Mendelson, and R. Ronen, "Filtering Techniques to Improve Trace Cache Efficiency," *Proc. 10th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '01)*, 2001.

[30] A. Ramirez, J.L. Larriba-Pey, and M. Valero, "Trace Cache Redundancy: Red & Blue Traces," *Proc. Sixth Int'l Conf. High Performance Computer Architecture*, 2000.

[31] A. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *IEEE TCCA Computer Architecture Letters*, vol. 1, 2002.

[32] D.A. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons," *Proc. Seventh Int'l Conf. High-Performance Computer Architecture (HPCA '01)*, 2001.

[33] D. Kaeli and P. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proc. 18th Int'l Symp. Computer Architecture (ISCA 91)*, 1991.

[34] O.J. Santana, A. Falcón, E. Fernández, P. Medina, A. Ramirez, and M. Valero, "A Comprehensive Analysis of Indirect Branch Prediction," *Proc. Fourth Int'l Symp. High Performance Computing*, 2002.

[35] 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.
[36] M.S. Hrishikesh, N.P. Jouppi, K.I. Farkas, D. Burger, S.W. Keckler, and P. Shivakumar, "The Optimal Logic Depth per Pipeline Stage Is 6 to 8 FO4 Inverter Delays," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '02),* 2002.
[37] T.M. Conte and S.W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* 1995.
[38] C.Y. Cher and T.N. Vijaykumar, "Skipper: A Microarchitecture for Exploiting Control-Flow Independence," *Proc. 34th Int'l Symp. Microarchitecture,* 2001.
[39] E. Rotenberg, Q. Jacobson, and J. Smith, "A Study of Control Independence in Superscalar Processors," *Proc. Fifth Int'l Conf. High-Performance Computer Architecture (HPCA '99),* 1999.
[40] H. Kim, O. Mutlu, J. Stark, and Y.N. Patt, "Wish Branches: Combining Conditional Branching and Predication for Adaptive Predicated Execution," *Proc. 38th Int'l Symp. Microarchitecture,* 2005.
[41] O.J. Santana, M. Galluzzi, A. Ramirez, and M. Valero, "An Analysis of Dynamic Instruction Streams," *Proc. XIV Spanish Workshop Parallelism,* 2003.
[42] P.J. Joseph and S. Vajapeyam, "Improving Control Flow Prediction by Exploiting Loop Constructs," Technical Report IISc-CSA-2000-5, Computer Science and Automation Dept., Indian Inst. of Science, 2000.
[43] T. Sherwood and B. Calder, "Loop Termination Prediction," *Proc. Third Int'l Symp. High Performance Computing (ISHPC '00),* 2000.
[44] M.R. de Alba and D.R. Kaeli, "Path-Based Hardware Loop Prediction," *Proc. Fourth Int'l Conf. Control, Virtual Instrumentation, and Digital Systems (CICINDI '02),* 2002.
[45] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson, "Power Awareness through Selective Dynamically Optimized Traces," *Proc. 31st Int'l Symp. Computer Architecture (ISCA '04),* 2004.

Oliverio J. Santana received the BSc and MSc degrees in computer science in 2000 from the University of Las Palmas de Gran Canaria (ULPGC), Canary Islands, Spain, and the PhD degree in 2005 from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain. He is an associate professor in the Computer Science and Systems Department at ULPGC. His research interests include complexity-effective fetch and decoding architectures, performance evaluation methodologies, and exploiting program semantic information. He is a coauthor of 20 international publications and is currently supervising two PhD students. He has served in the organization of several international conferences as a program committee member, an external reviewer, and submission Webmaster. He is a member of the IEEE.

Alex Ramirez received the BSc, MSc, and PhD degrees in computer science from the Universitat Politècnica de Catalunya in 1995, 1997, and 2002, respectively. He received the extraordinary award for the best PhD dissertation. He is an associate professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya and the leader of the Computer Architecture Group at the Barcelona Supercomputing Center. He was a summer student intern with Compaq's Western Research Laboratory (WRL), Palo Alto, California, from 1999 to 2000 and with Intel's Microprocessor Research Laboratory, Santa Clara, California, in 2001. His research interests include compiler optimizations, high-performance fetch architectures, multithreaded architectures, and vector architectures. He is a coauthor of more than 50 papers in international conference proceedings and journals and has supervised three PhD students. He is a member of the IEEE.

Mateo Valero has been a full professor in the Computer Architecture Department at the Universitat Politècnica de Catalunya (UPC) since 1983. Since May 2004, he has been the director of the Barcelona Supercomputing Center, National Center of Supercomputing, Spain. He is the coordinator of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). His research topics are centered in the area of high-performance computer architectures. He is a coauthor of more than 400 publications. He has served in the organization of more than 200 international conferences. He is a fellow of the IEEE and the ACM. He is an academic of the Royal Spanish Academy of Engineering, a correspondent academic of the Royal Spanish Academy of Sciences, and an academic of the Royal Academy of Science and Arts. His research has been recognized with several awards, including the 2007 Eckert-Mauchly Award, two National Awards on Informatics and on Engineering, and the Rey Jaime I Award in basic research. He received a Favourite Son Award from his hometown, Alfamén (Zaragoza), which named their public college after him.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.