Fetching instruction streams

Alex Ramirez, Oliverio J. Santana, Josep L. Larriba-Pey, Mateo Valero
Computer Architecture Department
Universitat Politecnica de Catalunya
{aramirez,osantana,larri,mateo}@ac.upc.es

Abstract

Fetch performance is a very important factor because it effectively limits the overall processor performance. However, there is little performance advantage in increasing front-end performance beyond what the back-end can consume. For each processor design, the target is to build the best possible fetch engine for the required performance level. A fetch engine will be better if it provides better performance, but also if it takes fewer resources, requires less chip area, or consumes less power.

In this paper we propose a novel fetch architecture based on the execution of long streams of sequential instructions, taking maximum advantage of code layout optimizations. We describe our architecture in detail, and show that it requires less complexity and resources than other high performance fetch architectures like the trace cache, while providing a high fetch performance suitable for wide-issue superscalar processors.

Our results show that using our fetch architecture and code layout optimizations obtains 10% higher performance than the EV8 fetch architecture, and 4% higher than the FTB architecture using state-of-the-art branch predictors, while being only 1.5% slower than the trace cache. Even in the absence of code layout optimizations, fetching instruction streams is still 10% faster than the EV8, and only 4% slower than the trace cache.

Fetching instruction streams effectively exploits the special characteristics of layout optimized codes to provide a high fetch performance, close to that of a trace cache, but has a much lower cost and complexity, similar to that of a basic block architecture.

1. Introduction

Fetch performance is a very important factor, because in a classic processor design, it is not possible to execute instructions faster than they can be fetched. In this sense, the fetch engine effectively limits the overall processor performance.

However, there is little performance advantage in increasing front-end performance beyond what the back-end can consume. For each processor design, the target is to build the best possible fetch engine for the required performance level. A fetch engine will be better if it provides higher performance, but also if it takes fewer resources, requires less chip area, or consumes less power.

The fetch engine has evolved from fetching a single instruction every few cycles, to one instruction per cycle, to a full basic block per cycle, to a full instruction trace per cycle. Each new architecture increases performance over the previous one, but also increases the cost and complexity.

The increase in complexity derives from the fact that the fetch engine has lost its relationship with the high-level programming constructs that it is fetching. While fetching basic blocks, the architecture is still aware of what it is fetching. When fetching instruction traces, or any other randomly constructed fetch structure, the architecture loses track of what the code is doing, and that increases complexity.

High fetch performance is always desirable, but to maintain complexity under control, it is necessary to design the fetch engine around the high-level programming constructs. This is what we have done with instruction streams.

An instruction stream is a sequential run of instructions, from the target of a taken branch, to the next taken branch (also called a *dynamic block* in the literature). A single instruction stream may contain multiple basic blocks, and multiple branches, as long as all the intermediate branches are not-taken.

As such, an instruction stream is fully identified by the starting instruction address and the stream length. Unlike traces, streams do not require information about the behavior of the branches contained in the stream, because it is implicit in the definition: all intermediate branches are not taken, and the terminating branch is always taken.

Furthermore, instruction streams are defined by the program (the branches, and the branch behavior) not by hardware limitations, and once again provide semantic information about the code behavior to the fetch engine

Figure 1 shows an example control flow graph from which we will find the possible streams. The figure shows

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

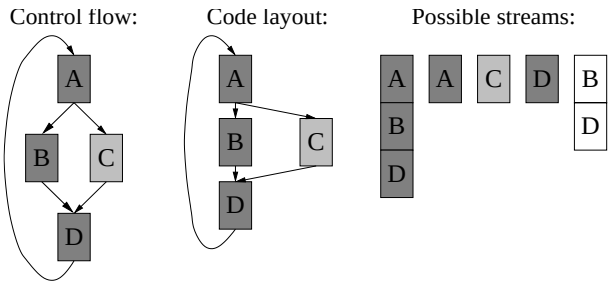

Figure 1. Example instruction streams.

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

From the resulting code layout we may encounter four possible streams composed by basic blocks ABD, A, C, 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. The infrequent case follows the taken branch at the end of A, goes through C, and jumps back into basic block D.

These are all the possible streams found as long as branch prediction is correct. In our architecture, instruction streams are not treated as atomic regions of code. That is, if a branch misprediction is detected halfway through a stream, the front-end engine is informed, fetch is redirected, and proceeds from the point of misprediction. We do not rollback execution to the beginning of the stream. Instructions up to the mispredicted branch are executed and graduated normally. The front-end recovers after the branch, and fetch is redirected towards the correct branch target.

If we account for branch mispredictions, there is a fifth possible stream which does not follow the definition above. Stream BD does not start at the target of a taken branch: it starts at the target of a branch misprediction. That is, if the branch at the end of A is predicted taken, but it turns out to be not taken, execution would continue from basic block B (we do not rollback to basic block A), but there is no full stream starting there.

To avoid this situation, we define a *partial* instruction stream as a stream which starts at the target of a branch misprediction, and goes on until the next taken branch. This allows the front-end engine to maintain the stream semantics even in the event of a branch misprediction.

Table 1 summarizes the reasons for designing a fetch engine around the concept of an instruction stream. Streams directly map to the structure of the high-level programming constructs, respecting loop bodies and hammock structures. Streams span multiple basic blocks and represent long sequences of instructions, specially in layout optimized codes, as we show in [28].

Fetch unit	high level	size (inst.)	cost	complex.	perform.
Basic block	yes	5–6	low	avg.	low
Trace cache	no	14	high	high	high
rePLay	no	60–100	high	high	high
Streams	**yes**	**20+**	**low**	**low**	**high**

Table 1. A comparison of fetch engines in terms of their relation to high-level code, cost, and performance.

Based on these facts, we will show that a fetch engine based on streams has a low implementation cost, results in a low complexity fetch architecture, and deliver high fetch performance comparable to that of a trace cache.

This paper describes in detail our implementation of the stream front-end architecture (a front-end valid for any back-end), which introduces a novel feature: the next stream predictor, which provides stream level sequencing (that is, it steps through the code one stream at a time), allowing the rest of the fetch engine to fetch multiple consecutive basic blocks in a single cycle without need of complex circuitry.

Our results show that indeed the stream fetch architecture provides for high performance, comparable to that of a trace cache, while maintaining a low implementation cost and complexity, similar to that of a basic block architecture.

In Section 2 we discuss previous related work, including state of the art fetch architectures like the FTB proposed by Reinman, Austin and Calder [30] and the trace cache architecture as proposed by Rotenberg, Bennett and Smith in [32].

In Section 3 we describe our proposed stream fetch architecture in detail, showing how it improves on other architectures by exploiting the special characteristics of layout optimized applications without requiring additional complexity.

In Section 4 we provide simulation results comparing our stream architecture with the Alpha EV8, the FTB, and the trace cache architectures to back up our claims about the stream front-end architecture:

- Fetching streams provides higher performance than the EV8 and FTB front-ends, and a performance close to that of a trace cache, maintaining a reduced cost and minimum complexity.
- Even in the absence of code layout optimizations, the stream fetch architecture still provides performance improvements over the EV8 and FTB architectures.

Finally, in Section 5 we provide our conclusions for this work, and provide guidelines for future development of this architecture.

2. Related work

Superscalar processors require a fetch engine capable of providing more than one instruction per cycle in order to keep their functional units busy. However, fetching more instructions per cycle can not be accomplished by replicating the fetch engine.

2.1. The FTB front-end

In order to increase fetch width, Yeh and Patt introduced a novel fetch architecture based on dynamic branch prediction [39]. Figure 2 shows a block diagram of this fetch architecture.

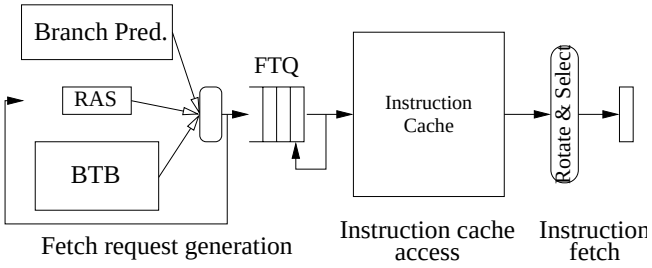

Figure 2. The BTB fetch architecture uses dynamic branch prediction to fetch a full basic block per cycle.

The fetch engine is divided in two sections: the dynamic branch prediction mechanism, composed of the BTB, the return address stack (RAS), and the two-level branch predictor (BP); and the instruction cache. The BTB provides information about the basic block being fetched, and the BP predicts the behavior of its terminating branch. All the provided data is combined to calculate the fetch address for the next basic block. The same information drives the instruction cache to provide a cache line, and select the valid instructions from it.

The original design by Yeh and Patt [39] had a single table for the BTB and branch predictor. Calder and Grunwald [3] showed that decoupling the BTB and branch predictor allowed an independent resource allocation which leads to better branch prediction accuracy. They also suggested that basic blocks should not be terminated at branches which have never been taken, that is, that only taken branches should introduce a basic block in the BTB.

Further development of this fetch architecture leads to a decoupling of the dynamic branch prediction mechanism and the instruction cache access, as proposed by Reinman, Austin, and Calder [30]. The branch prediction mechanism is a fully autonomous engine, capable of following a speculative path without further assistance. Each cycle it generates the fetch address for the next cycle, and a fetch request which is stored in a fetch target queue (FTQ). The instruction cache is then driven by the requests stored in the FTQ.

Another important contribution of [30] is the Fetch Target Buffer (FTB). It extends the BTB by allowing the storage of variable length *fetch blocks*. A fetch block is a sequence of instructions starting at a branch target, and ending at a strongly biased taken branch. This allows strongly biased not taken branches to be embedded within a fetch block, increasing the fetch width without increasing the cost, as such not taken branches can be easily predicted by simply ignoring them.

Code layout optimizations benefit this fetch architecture in two ways: reducing the number of instruction cache misses, and increasing the effective fetch width. Because branches are aligned towards their not taken direction, it is likely that a branch which exhibits a biased behavior will be biased towards not taken. If a branch is always not taken, it is effectively ignored by the FTB architecture, and it never terminates a fetch block, enlarging the size of the fetch unit of the architecture.

The stream fetch architecture takes this advantage one step further. The FTB ignores *biased* not taken branches, but the next stream predictor ignores *all* not taken branch instances. For example: if a branch is 100% not taken, it will be ignored by both the FTB and the next stream predictor. If a branch is only 80% not taken, it will be ignored by the FTB until it is taken for the first time. After it has been taken once, it enters the FTB tables, and always terminates the fetch block.

The FTB architecture does not store overlapping fetch blocks. If a taken branch is found half-way through a fetch block, the fetch block is split in two smaller parts. Meanwhile, the stream fetch architecture allows for overlapping fetch blocks (streams), choosing the appropriate one fore each instance. This allows the stream predictor to ignore the branch in all its not taken instances, which allows a larger basic block to be fetched 80% of the time for than particular branch.

2.2. The trace cache

Fetching one basic block per cycle effectively limits the fetch performance to 5–6 instructions per cycle on most integer benchmarks. In order to feed an 8 or 16 wide superscalar processor, fetching multiple basic blocks per cycle becomes necessary.

The trace cache [9, 23, 31, 32] is one such high fetch width mechanism, recently implemented in the Pentium4 processor[14][1]. Figure 3 shows a block diagram of the

[1]This shows that regardless of its high cost and complexity, the trace cache mechanism is indeed implementable. Our work does not suggest that

trace cache mechanism as proposed by Rotenberg, Benett and Smith in [32].

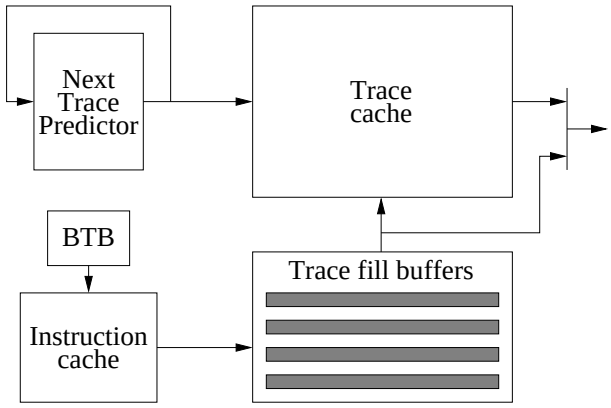

Figure 3. The trace cache fetch architecture.

The trace cache captures the dynamic instruction stream, and fragments it in smaller segments called traces. These traces are then stored in a special purpose cache (the trace cache), expecting that the same code sequence will be re-executed in the future. If such is the case, it can be provided from the trace cache in a single cycle without need of further processing regardless of taken branches.

A trace is defined by heuristics which limit either the number of instructions in a trace, the number of branches, the number of indirect branches, and such. In order to fully identify an instruction trace, it is necessary to know the starting instruction of the trace, and the number and behavior of the branches contained. A trace may contain multiple basic blocks, and several branches, regardless of them being taken or not taken.

The next trace predictor [17] provides trace level sequencing. That is, the fetch engine steps through the code at the trace level granularity. However, when a branch misprediction is encountered, or the predicted trace is not found in the trace cache, instructions must be provided by a secondary fetch engine. The secondary fetch engine is composed by an instruction cache and a BTB for dynamic branch prediction.

There has been much work done to improve the trace cache mechanism. Techniques such as path associativity, partial matching, and inactive instruction issue [9] target an improvement in the trace cache hit rate. Branch promotion [21] targets an increase in the average trace length, allowing traces to contain more branches. The possibility of using the trace buffers to do dynamic code optimization has also been explored [10].

building a trace cache is not feasible, we just propose an easier alternative with similar performance.

2.3. Other high-performance fetch architectures

Aside from the widely adopted trace cache, there are other high performance fetch mechanisms, with varying degrees of complexity and performance. The branch address cache [38], and the collapsing buffer [7] represent earlier attempts at a fetch architecture capable of fetching multiple non-sequential basic blocks in a single cycle. Both require multiple read ports to the instruction cache, a complex branch predictor, and a complicated realignment network to join the fetched blocks before passing them to the decode stage.

Other multiple branch predictors like the multiple block-ahead predictor [35], or the tree-like subgraph predictor [8] can also be used to implement a fetch engine capable of providing multiple basic blocks per cycle, but would also require a high complexity alignment network. The trace cache solves this problem by moving the alignment network out of the critical path, and storing the already aligned instructions in a special purpose cache.

The next line and set predictor (NLS) architecture [5], implemented in the Alpha 21264 [12], also allows fetching of multiple basic blocks in a single cycle, as long as they reside sequentially in the same cache line.

The Alpha EV8 architecture [34] uses an interleaved BTB and a multiple branch predictor to fetch instructions from multiple basic blocks up to the first encountered taken branch, much in the way the SEQ.3 engine described in [31].

The rePLay microarchitecture [22] uses a front end derived from the trace cache, making extensive use of the branch promotion technique to build very long instruction traces, called frames, and then dynamically optimize them. Opposite to streams, frames are defined as atomic regions. That is, if a branch misprediction is detected halfway through a frame, execution is backed up to the beginning of the frame, and execution proceeds again fetching instructions from the instruction cache instead, one basic block at a time.

The stream architecture uses the static approach to solve the same problems. Both the trace cache and rePLay read the dynamic instruction stream and record segments of it for increased fetch performance or dynamic optimization, and to avoid the alignment network required by other mechanisms. We rely on the compiler to organize the code so that the code segments that would be constructed by the trace cache/rePLay are already mapped sequentially in memory. This allows the stream architecture to use the instruction cache as the only source for instructions, avoiding any redundant and/or complex mechanisms.

Finally, superblocks bear a non-casual resemblance to instruction streams, and they are also composed of multiple sequentially executed basic blocks, and are also organized by the compiler. IBM holds a later patent on a branch

predictor designed to fetch (and prefetch) superblocks [20]. While our mechanism pursues a similar goal (fetching long sequences of sequential instructions), we achieve it in a different way. A detailed comparison between both can be found in [33].

A performance comparison between the stream fetch architecture and all other high-performance architectures is beyond the scope of this work. Instead we have chosen to compare with the state of the art regarding fetch architectures for sequential basic blocks (the FTB and the EV8 architectures), and the most widely adopted mechanism for non-sequential basic blocks (the trace cache).

2.4. Code layout optimizations

Previous work has explored the use of code layout optimizations to increase fetch performance both in BTB architectures and trace cache architectures. Code layout optimizations were initially proposed to improve the performance of the instruction memory hierarchy (instruction cache, instruction TLB) by reducing the code footprint and minimizing conflict misses [15, 24, 37, 13, 11], and to align branches to benefit the underlying fetch architecture and branch predictor [4]. Our previous work presents a detailed analysis [25] of the effects of these optimizations, concluding that the improvements on the instruction cache performance are due to an increase in the sequential execution of code, and a better packing of useful code to cache lines.

We also show that layout optimizations not only improve instruction memory performance, but also have an impact on the branch prediction accuracy [27], and the effective fetch width of the front-end [26]. The layout optimizations align branches towards not-taken, which translates in longer chains of sequential instructions being executed.

The use of code layout optimizations usually has the availability of profile data as a requisite, and experience shows that it is infrequent for end-users to do so. In those cases in which profile data is unavailable, it is still possible to benefit from code layout optimizations either using heuristics to replace the profile data [2], or dynamic code optimizers [1, 19].

While our stream fetch architecture is designed to exploit the special characteristics of optimized codes, it does not exclusively depends on them. However, the use of such heuristics or dynamic optimizations would provide an additional fetch performance increase due to the synergy of the layout optimizations and our architecture.

3. The stream fetch architecture

Our previous work with the *Software Trace Cache* [26] shows that a sequential fetch engine obtains a fetch performance similar to that of a trace cache when using layout optimized codes. The sequential engine used in that work is the SEQ.3 unit described in [31]. However, such engine proves very complex to implement, and is still limited to 3 consecutive basic blocks per cycle. Next, we describe a fetch engine designed to fetch sequential code, up to a whole instruction stream per cycle, that overcomes the 3 basic block limitation without requiring more complexity than the classic BTB front-end engine.

Figure 4 shows the block diagram for the proposed stream fetch engine. The next stream predictor provides the fetch engine with stream level sequencing. That is, given the current stream starting address, it provides the current stream length, and the next stream starting address. The predicted next stream address is used as the fetch address for the next cycle. The current stream address, and the current stream length are stored in the fetch target queue (FTQ), and represent a fetch request for a full instruction stream.

The instruction cache is driven by the fetch requests stored in the FTQ. The stream starting address is used to access the instruction cache, which provides one or more consecutive cache lines. If the cache lines provided contain the whole stream, the FTQ is advanced to the next request, if not, the fetch request is updated to reflect the remaining part of the stream to be fetched.

As described, the stream fetch engine can be run as a single pipeline stage, or can be divided in several shorter stages using the different intermediate structures (FTQ and cache line buffer) to separate them. In the following sections we describe the different fetch sub-stages in detail.

The novel aspect of this fetch architecture is the use of a specialized branch predictor (the stream predictor) which provides stream level sequencing. Each prediction contains information about a whole instruction stream, possibly containing multiple basic blocks.

The use of an FTQ is not novel, it was introduced in [30]. It decouples the branch prediction from the memory access, and stores information about the instruction sequence as indicated by the branch predictor providing a certain degree of tolerance to the branch predictor latency, and storing a glimpse of the future control flow of the program. In our case, the usefulness of the FTQ increases, as each entry now contains information about a whole instruction stream instead of just a fetch block.

Our fetch architecture has a single source for instructions (the instruction cache), which uses very wide lines to provide a high fetch width. Fetching streams does not require any additional instruction storage not special purpose caches.

3.1. Complexity issues

Across al the paper we claim that the stream fetch architecture has a lower cost and complexity than a trace cache architecture, but quantifying the cost of both architectures to allow an objective comparison is beyond the scope of this work.

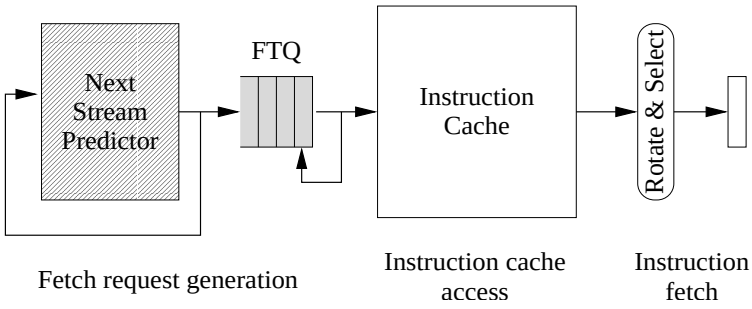

Figure 4. The proposed stream fetch engine. The instruction stream becomes the basic fetch entity.

However, it is clear that the trace cache requires two separate instruction paths (a primary path for traces, and a secondary path when the trace cache misses), requiring additional storage, and redundant branch prediction.

The stream fetch architecture does not require additional instruction storage (the trace cache), nor a trace construction engine (streams are recorded, not built dynamically), nor two separate branch predictors (a trace predictor, and a back-up predictor for trace construction).

The trace cache uses a BTB as a back-up predictor in case of trace predictor misses, which makes it necessary to keep the BTB updated at any point, requiring an update for each executed branch. The BTB is the backup predictor for the trace cache, and is used on a trace predictor miss, or when a misprediction is detected half-way through a trace. In the event of a stream predictor miss, we resort to sequential fetching, which does not require any back-up predictor.

Our stream fetch architecture has a single instruction path (as opposed to two in the trace cache), a single branch predictor, a single storage cache, and is fairly straightforward to implement, without need of much control logic to coordinate the different components, as they all work decoupled from each other by the FTQ.

In any case, we do not wish to disregard other advantages of the trace cache mechanism, like its ability to store decoded instructions for processors like the Pentium4 [14], or enabling dynamic optimization of traces [9, 22]. Our instruction stream fetch architecture is intended as another alternative for a high-performance fetch architecture when complexity and cost issues are to be involved, but does not completely replace the trace cache in all its possible uses.

3.2. Stream prediction

The stream fetch engine we propose is driven by a branch predictor providing stream level sequencing. Such is the purpose of the *next stream predictor.* Given a stream starting address, the predictor provides a stream identifier and the next stream starting address. The stream identifier consists of the stream starting address and the stream length.

The next stream predictor serves both the purpose of branch direction predictor and target address predictor, replacing both the conditional branch predictor and the BTB/FTB of a conventional fetch engine. The branch directions are predicted implicitly: all intermediate branches from the start address of the stream are predicted not taken, and the stream terminating branch is predicted taken. The target addresses for all not taken branches are the next sequential instruction, and the target address of the terminating branch is the starting address for the next stream.

Figure 5 shows our implementation of the next stream predictor (a cascaded next stream predictor). Each table entry contains information about one stream: start address, length (in instructions or bytes in a variable-length ISA), terminating branch type (for return stack management), the next stream address, and a 2-bit saturating counter used for the replacement policy.

The first table is indexed using the current fetch address only, and the second table is indexed using a hash of the current fetch address and the previous stream starting addresses (the previous fetch addresses). The hash function uses a DOLC scheme similar to what was used in multiscalar processors [16].

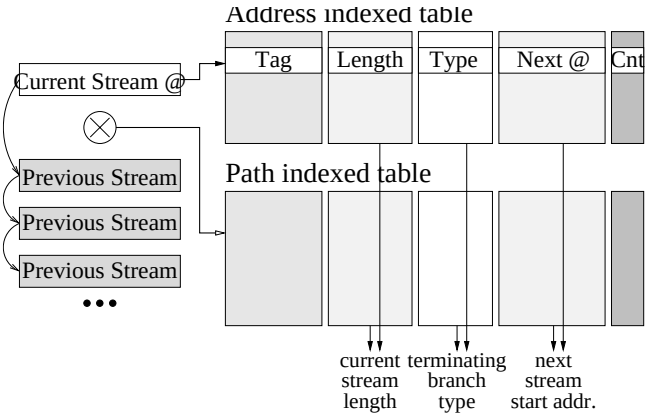

Figure 5. Cascaded implementation of the next stream predictor.

The predictor maintains two separate path history registers: a *lookup* register which is updated immediately with

speculative information, and an *update* register which is updated at commit time when the stream has completed, using correct path information only. In the case of a misprediction, the contents of the non-speculative register is copied to the speculative register, restoring the correct history state.

To improve the prediction accuracy of return instructions, we use a Return Address Stack (RAS). The RAS is updated speculatively as guided by the branch type field, and a shadow copy of the top of the stack is kept with each branch instruction. When a misprediction is detected, the stack index and the top of the stack are restored to their correct values.

To obtain a prediction, the stream predictor is presented with the current fetch address (the address where the new stream starts). The predictor calculates the hash function, and obtains an index into the prediction tables. If both tables have a hit, we choose the data from the path correlated table. If only one hits, we use whichever data is provided. If both tables miss, we resort to sequential fetching until the predictor hits again, or a branch misprediction is detected.

The confidence counter is used to implement the replacement policy. When a new stream is completed, the prediction tables are checked. If the stream is already there (the table is being updated with the same length and target address already stored in that entry) the confidence counter is increased. If the new stream data and the data stored in the table do not match (either the stream length or target address differ), the counter is decreased. When the counter reaches zero, the old data is replaced by the new data (both length and target are replaced) and the counter is set to one.

The use of path correlation and the hysteresis counter for replacement is what allows the stream predictor to hold overlapping streams in the prediction tables. This allows the streams to be kept as long as possible, without having to cut them short to avoid overlapping of multiple fetch blocks.

A stream is introduced in both tables the first time it appears. In following appearances of the stream, its information is updated only in those tables where it has not yet been replaced. Streams which do not require patch correlation for accurate prediction will be replaced from the second table, but the first table will still be able to predict them.

A stream present only in the first table will be upgraded to the second table if it is mispredicted. Streams not requiring path correlation will thus never be upgraded to the second table, avoiding aliasing.

Each access to the predictor provides information about a whole instruction stream, possibly containing multiple basic blocks as long as they are connected by not taken branches. This is how the stream front end engine takes advantage of the characteristics of layout optimized codes: an average of 80% of all conditional branch instances are not taken, while only 60% of all branches are strongly biased to not taken. By stepping through the code passing through not taken branches, we are able to fetch much longer code sequences than if we had to stop at all branches to predict them individually. And by ignoring not taken branch instances, we avoid interference in the prediction tables, which allows us to increase prediction accuracy.

3.3. Fetch target queue

Following the proposal of Reinman, Austin and Calder [30] we have decoupled the branch prediction stage from the instruction cache access stage. The stream predictor provides information about an instruction stream with each prediction, the predictions are stored in a Fetch Target Queue (FTQ), and are used to drive the instruction cache access.

The FTQ does not serve any actual performance improvement purpose, the branch prediction and instruction cache access could be done in parallel and avoid the use of an FTQ and the additional pipeline stage. We have used it because it allows the branch predictor to work at a different rate than the instruction cache. It is likely that a single FTQ entry will take multiple cycles to fetch from the instruction cache (long basic blocks, or streams containing multiple basic blocks). This allows the branch predictor to run ahead of the instruction cache, and to stall (and avoid power consumption) when the FTQ is full[2].

The usefulness of the FTQ with our stream front-end is higher than that of a basic block architecture, because each FTQ entry contains information about more instructions, which allows the front-end to have a larger view of the future instruction stream.

The average stream contains over 16 instructions, which means that the average fetch request stored in the FTQ is too large to be fetched in a single cycle. Instead of dividing a large fetch request into several smaller ones, we have implemented a fetch request update mechanism as shown in Figure 6.

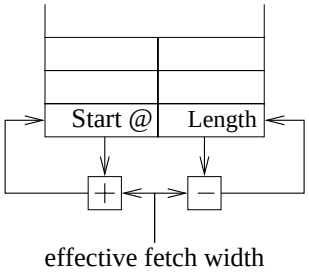

Figure 6. Fetch target queue update mechanism.

[2]This is not the case of the trace cache, where each trace prediction corresponds to a single trace cache access. In the trace cache setup, if the fetched trace can feed the processor for more than one cycle, both the predictor and the trace cache stall at the same time.

Each fetch request contains the stream starting address and the stream length. The starting address is used to access the instruction cache and fetch one or more consecutive cache lines. Depending on how many cache lines were fetched, and the cache line width, a number of instructions belonging to the stream will be fetched.

The fetch request is updated using the actual number of instructions obtained from the instruction cache. The stream starting address is advanced, and the stream length is reduced appropriately. If the stream length reaches zero, then the fetch request has been satisfied, and the FTQ is advanced to the next request.

3.4. Instruction cache

Instruction streams are composed of sequential instructions. The easiest way to fetch an instruction stream is to read multiple consecutive cache lines from the instruction cache until the whole stream has been fetched. However, fetching a large number of cache lines in a single cycle is not always feasible, nor cost-effective.

The simplest fetch mechanism would be to fetch a single cache line per cycle. In this scenario we must face the problem of instruction misalignment, as shown in Figure 7.

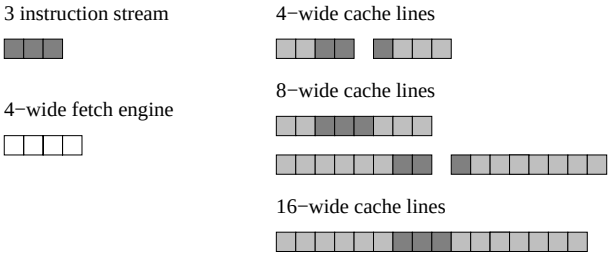

Figure 7. The instruction misalignment problem.

A fetch request consisting of 3 consecutive instructions should be fetched in a single cycle by a 4-wide fetch engine, but such is not always the case. It is possible that the 3 instruction stream is split across two separate cache lines, which means that it will take two cycles to fetch if we fetch a single cache line per cycle.

The use of long instruction cache lines alleviates this problem. A longer cache line reduces the possibilities of the instruction stream crossing the cache line boundary.

Also, previous work shows that layout optimized codes benefit from long cache lines more than unoptimized codes due to a denser packing of useful instructions to cache lines. These results show that even a very long 128-byte line (32 instructions) is usually fully used before being replaced [25].

Considering both benefits together (the reduced stream misalignment, and the instruction cache miss rate benefits), we have adopted a simple instruction cache design which reads a single line per cycle, but we use a very long line size.

An alternate solution would be to fetch two consecutive cache lines from a multi-banked instruction cache, so that we can always guarantee a full width of instructions, as done in [7, 31]. Our solution requires a wider read port to the instruction cache, while this solution requires an interchange network which increases complexity. For the remaining of this paper we will use the wide cache line approach.

4. Performance evaluation

4.1. Simulation setup

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

We compare our stream fetch architecture with three other state-of-the-art fetch architectures: the FTB architecture [30] using a perceptron branch predictor [18], the Alpha EV8 architecture using a 2bcgskew predictor [34], and the trace cache architecture using a trace predictor [32] and selective trace storage [29].

Table 2 shows the values used in the processor simulation. Most of the setups correspond to the fetch engine, which was simulated in greater detail.

In all cases in which the branch predictor requires two separate mechanisms (branch predictor and target address predictor) we have simulated multiple resource allocations maintaining a total approximate budget of 45KB, and include here only the one which provided better performance.

The trace cache processor uses a 32KB trace cache (instruction storage only), and a 32KB instruction cache, both 2-way set associative. We tested multiple combinations of instruction and trace cache sizes (big trace cache with small instruction cache and vice versa), and multiple secondary fetch paths (multiple block instruction cache with a multiple branch predictor, single block instruction cache with a classic branch predictor), and selected the one which provided better performance. For the trace cache, we use only the trace packing and selective trace storage optimizations[3].

The results shown are the harmonic average of all SPEC 2000 integer benchmarks. We feed our simulator with traces of 300 million instructions collected using the *ref*

[3]Our results show that in the context of code layout optimizations, the partial matching optimization actually causes a drop in trace cache performance.

FTB architecture + perceptron	
history	512 perceptrons 40 bit global history 4096 x 14 bit local history
FTB	2048-entry, 4-way
RAS	8-entry
EV8 fetch architecture + 2bcgskew	
history	4 x 32K-entry tables 15 bit history
BTB	2048-entry, 4-way
RAS	8-entry
Stream fetch architecture	
first table	1K-entry, 4-way
second table	6K-entry, 3-way
DOLC index	12-2-4-10
RAS	8-entry
Trace cache architecture + Trace predictor	
fist level	1K-entry, 4-way
second level	4K-entry, 4-way
DOLC index	9-4-7-9
RHS	8-entry
Backup BTB	1K-entry, 4-way
Trace cache	32KB, 2-way, Selective Trace Storage
Common settings	
pipe width	2, 4, and 8
pipe depth	16 stages
FTQ	4 entries
L1 inst. cache	64KB, 2-way, 1-cycle (single ported)
L1 inst. line	4x pipe width (32, 64, and 128-bytes)
L1 data cache	64KB, 2-way, 64B line, 1-cycle
L2 cache (unif.)	1MB, 4-way, 64B line, 15-cycles
Memory	100-cycles latency

Table 2. Processor setup simulated.

input set. To find the most representative instruction segment we have analyzed the distribution of basic blocks as described in [36].

Previous has shown that code layout optimizations have a very important effect on all aspects of fetch performance. For this reason we use two separate sets of executables: the baseline set, and a layout optimized set. The layout optimized codes were generated with the *spike* tool shipped with Compaq Tru64 Unix 5.1 [6]. The profile data was obtained using *pixie* and the *train* input set. As mentioned above, we use the *ref* input set to obtain simulation results. It is important to note that we used both optimized and unoptimized codes with all fetch architectures.

4.2. Processor performance

In Section 3 we have described the stream fetch architecture: a low cost, low complexity fetch engine designed to exploit the characteristics of optimized code layouts. As we already mentioned in the introduction, a fetch engine is better than others if it has a lower cost, requires fewer resources, or has a lower energy consumption. However, raw performance is still a very important metric when designing a high performance processor.

Figure 8 compares the processor performance for the different fetch architectures varying the pipeline width, using both baseline and optimized code layouts.

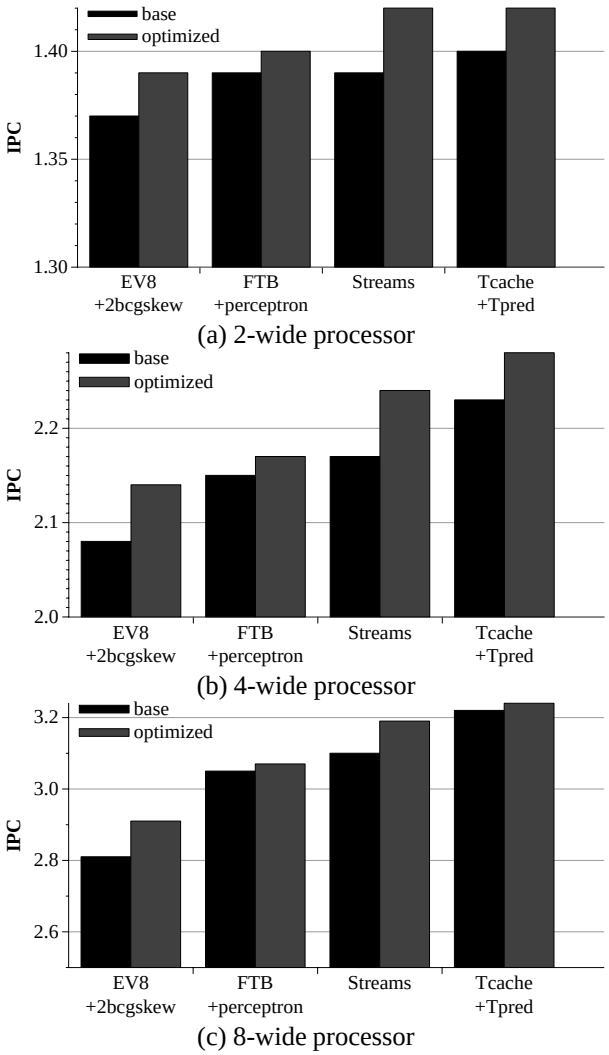

Figure 8. IPC performance for multiple pipeline widths using base and optimized codes.

The results for the 2-wide processor show that indeed there is very little performance advantage in increasing fetch performance beyond what the back-end can consume. In this narrow pipeline all setups obtain a similar performance, with the streams and the trace cache being only 2% better than the EV8 architecture.

Given this circumstance, the best option would be the one which obtains that performance at the minimum cost and complexity, or consumes the less power. Our fetch architecture has a slightly higher performance, and does not introduce additional costs and complexity, which make it the more suitable choice.

The 4-wide pipeline already shows significant performance differences, as the fetch width and branch prediction accuracy become more important. The results show that the use of code layout optimizations is beneficial to all architectures, but it is the stream fetch architecture which benefits most from them.

Using optimized codes, the stream fetch architecture obtains a 5% speedup compared to the EV8 architecture, and a 3% speedup agains the FTB, while being only 2% slower than the trace cache architecture.

Even in the absence of layout optimized codes the stream fetch architecture still obtains a 4% speedup against the EV8 architecture, and a 1% speedup against the FTB. That is, the streams architecture using baseline codes obtains the same performance as the FTB architecture using optimized applications.

Also, it is worth noting that the combination of optimized applications and the stream fetch architecture obtains better performance than the trace cache using unoptimized codes (this result is also visible in the 2-wide pipeline). The results in Table 3 will show that this is due to the higher prediction accuracy of the stream predictor in an environment where the fetch width is still not the most relevant factor.

The 8-wide pipeline shows the larger performance differences, because in this wide pipeline setup, the fetch width becomes a more relevant aspect, as fetching instructions from multiple basic blocks per cycle becomes imperative.

When using unoptimized codes, both the FTB and stream fetch architectures provide an intermediate performance between the EV8 and the trace cache architectures (10% faster than EV8, and only 4–5% slower than the trace cache).

However, the use of code layout optimizations uncovers the potential performance of instruction streams. The stream fetch architecture is designed to exploit optimized codes, and does so successfully, increasing performance by a full 3%, while the FTB and the trace cache improve less than 1%.

When using code layout optimizations, the stream fetch architecture obtains a 4% speedup over the FTB, and is only 1.5% slower than the trace cache, while requiring a lower cost and complexity.

Figure 9 shows individual benchmark results for the 8-wide processor in the presence of code layout optimizations. The results show that the stream fetch architecture obtains the best performance in 5 benchmarks, (176, 186, 254, 255, and 256) and is at least second best is all but one case (175) where it is third best. Comparing only streams vs. traces performance, the trace cache is better in 6 codes, and the streams are better in 5 codes.

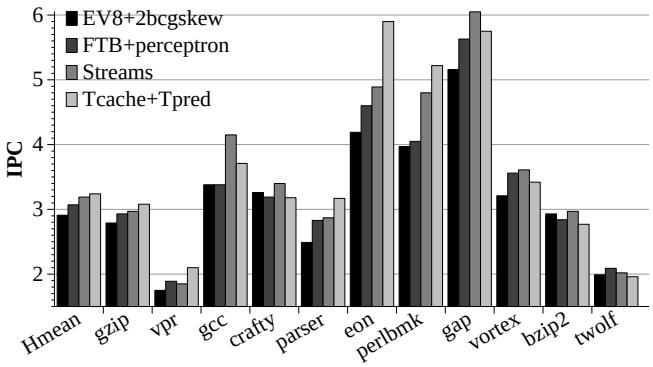

Figure 9. Individual IPC results for the 8-wide processor using optimized codes.

4.3. Fetch performance

Table 3 shows other performance metrics which determine the fetch engine performance: the actual fetch width on the 8-wide processor, and the branch misprediction rate. The instruction cache miss rate is very low in all setups, and hardly contributes to the IPC performance difference.

	base		optimized	
	Mispred.	Fetch IPC	Mispred.	Fetch IPC
EV8+2bcgskew	3.9%	5.3	3.9%	5.5
FTB+perceptron	3.0%	5.5	2.8%	5.5
Streams	2.5%	5.8	2.3%	6.0
Traces	2.8%	6.8	2.8%	6.9

Table 3. Branch misprediction rate and Fetch IPC for the 8-wide processor.

The results show that the main advantage of the trace cache over the other fetch architectures is the increased fetch width, obtained fetching instructions from multiple non-sequential basic blocks in a single cycle. This fetch width increase means that the trace cache provides 25–30% more instructions per cycle than the EV8 or FTB architectures. However, the stream fetch architecture does not lag behind, providing only 11–15% fewer instructions per cycle than the trace cache.

However, an increased fetch width is not the only advantage of the stream architecture. Our results also show that instruction streams also provide for increased prediction accuracy, with a misprediction rate 18% lower than that of the perceptron or trace predictors, and 36–40% lower than the 2bcgskew miss rate.

Instruction streams seem to be inherently easier to predict. The same seems to hold true for fetch blocks (FTB units) and traces (trace predictor units). As the prediction unit size increases, fewer predictions are required, which means that there is less pressure and interference in the pre-

diction tables. This would point the advantage to traces and streams, but traces are built using hardware-derived heuristics, and result in a somewhat random splitting of code. Meanwhile, streams are defined by the natural control flow of the program, retaining their relationship to high-level programming constructs, which makes them easier to predict.

The combination of increased fetch width and improved branch prediction accuracy explain the IPC improvements obtained fetching instruction streams. The fact that the trace cache provides only 11–15% more instructions per fetch than the stream architecture, and that the trace predictor has a misprediction rate 17% higher than the stream predictor show why the stream architecture comes so close to the trace cache performance.

5. Conclusions

This paper presents the instruction stream fetch architecture, our first step towards an architecture which ties the internal processing units to the high-level programming language structures.

Instruction streams are sequences of sequential instructions between two consecutive taken branches. As such, streams contain instructions from multiple basic blocks, but still maintain their relationship with high-level structures such as loop bodies. Their large size and their high-level information make them suitable for a low complexity, but high performance fetch engine.

Our fetch architecture replaces the conventional branch predictor with a next stream predictor, which steps through the code at a larger granularity (that of instruction streams), providing higher prediction accuracy and larger fetch blocks. Because streams are sequentially stored in memory, we use the instruction cache as our only source of instructions, avoiding redundant mechanisms and additional complexity.

Our results show that for narrow issue processors, the stream architecture offers higher performance than other state-of-the-art architectures, and a lower complexity than a trace cache. On wide issue processors, the use of code layout optimizations and our fetch architecture obtains 10% higher performance than the EV8 fetch architecture, and 4% higher than the FTB architecture using state-of-the-art branch predictors, while being only 1.5% slower than the trace cache.

When not using optimized applications, fetching instruction streams is still 10% faster than the EV8 fetch architecture, and only 4% slower than fetching traces, and always at a lower cost and complexity.

If performance is the only relevant factor, then the trace cache is still probably the best option. If implementation cost and complexity are also taken into account, we believe that our stream fetch architecture provides a better alternative.

Future lines of research include improving the accuracy of the next stream predictor, devising better ways to explode the characteristics of optimized codes, and further development of the instruction stream architecture.

Acknowledgments

This work was supported by the Ministry of Education and Science of Spain under contract TIC-2001-0995-C02-01, Generalitat de Catalunya under grant 2001FI-00724-APTIND, CEPBA, and an Intel Fellowship. Thanks go to Ayose Falcon for all his inputs, and his work on the simulation tool. The authors also want to thank the anonymous reviewers for their constructive inputs, and specially Brad Calder for his help in writing the final version of this paper.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2000.

[2] T. Ball and J. R. Larus. Branch prediction for free. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 300–313, June 1993.

[3] B. Calder and D. Grunwald. Fast & accurate instruction fetch and branch prediction. *Proceedings of the 21st Annual Intl. Symposium on Computer Architecture*, pages 2–11, 1994.

[4] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *Proceedings of the 6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, Oct. 1994.

[5] B. Calder and D. Grunwald. Next cache line and set prediction. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, June 1995.

[6] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: an optimizer for alpha/nt executables. *USENIX*, pages 17–23, Aug. 1997.

[7] T. Conte, K. Menezes, P. Mills, and B. Patell. Optimization of instruction fetch mechanism for high issue rates. *Proceedings of the 22th Annual Intl. Symposium on Computer Architecture*, pages 333–344, June 1995.

[8] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. *Proceedings of the 28th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 258–263, Nov. 1995.

[9] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 1997.

[10] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimization for trace cache microprocessors. *Proceedings of the 31st Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 173–181, Nov. 1998.

[11] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. *Proceedings of the 30th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, Dec. 1997.

[12] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, Oct. 1996.

[13] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 171–182, June 1997.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[15] W.-M. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*, pages 242–251, June 1989.

[16] Q. Jacobson, S. Bennett, N. Sharms, and J. Smith. Control flow speculation in multiscalar processors. *Proceedings of the 3rd Intl. Conference on High Performance Computer Architecture*, pages 218–229, Feb. 1997.

[17] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. *Proceedings of the 30th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, Dec. 1997.

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

[19] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, pages 136–147, May 1999.

[20] R. K. Nair. Method and apparatus for prefetching superblocks in a computer processing system. *US Patent 6,304,962 B1*, Oct. 2001.

[21] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *Proceedings of the 25th Annual Intl. Symposium on Computer Architecture*, pages 262–271, June 1998.

[22] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. *Proceedings of the 33rd Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 303–313, Dec. 2000.

[23] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. *U.S. Patent Number 5.381.533*, Jan. 1995.

[24] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 16–27, June 1990.

[25] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J. L. Larriba-Pey, G. Lawney, and M. Valero. Code layout optimizations for transaction processing workloads. *Proceedings of the 28th Annual Intl. Symposium on Computer Architecture*, July 2001.

[26] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *Proceedings of the 13th Intl. Conference on Supercomputing*, June 1999.

[27] A. Ramirez, J. L. Larriba-Pey, and M. Valero. The effect of code reordering on branch prediction. *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 189–198, Oct. 2000.

[28] A. Ramirez, J. L. Larriba-Pey, and M. Valero. A stream processor fron-end. *IEEE TCCA Newsletter*, pages 10–13, June 2000.

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

[30] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. *Proceedings of the 26th Annual Intl. Symposium on Computer Architecture*, pages 234–245, May 1999.

[31] E. Rotenberg, S. Benett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *Proceedings of the 29th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 24–34, Dec. 1996.

[32] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers, Special Issue on Cache Memory*, Feb. 1999.

[33] O. J. Santana, A. Falcon, A. Ramirez, J. L. Larriba-Pey, and M. Valero. Differences between the next stream predictor and the apparatus for prefetching superblocks described in us patent 6,304,962 b1. Technical Report UPC-DAC-2002-18, Universitat Politecnica de Catalunya, May 2002.

[34] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. *Proceedings of the 29th Annual Intl. Symposium on Computer Architecture*, 2002.

[35] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *Proceedings of the 7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

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

[37] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. *Proceedings of the 1st Intl. Conference on High Performance Computer Architecture*, pages 360–369, Jan. 1995.

[38] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *Proceedings of the 7th Intl. Conference on Supercomputing*, pages 67–76, July 1993.

[39] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *Proceedings of the 25th Annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 129–139, Dec. 1992.