DIA: A Complexity-Effective Decoding Architecture

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

Abstract—Fast instruction decoding is a true challenge for the design of CISC microprocessors implementing variable-length instructions. A well-known solution to overcome this problem is caching decoded instructions in a hardware buffer. Fetching already decoded instructions avoids the need for decoding them again, improving processor performance. However, introducing such special-purpose storage in the processor design involves an important increase in the fetch architecture complexity. In this paper, we propose a novel decoding architecture that reduces the fetch engine implementation cost. Instead of using a special-purpose hardware buffer, our proposal stores frequently decoded instructions in the memory hierarchy. The address where the decoded instructions are stored is kept in the branch prediction mechanism, enabling it to guide our decoding architecture. This makes it possible for the processor front end to fetch already decoded instructions from the memory instead of the original nondecoded instructions. Our results show that using our decoding architecture, a state-of-the-art superscalar processor achieves competitive performance improvements, while requiring less chip area and energy consumption in the fetch architecture than a hardware code caching mechanism.

Index Terms—Superscalar processor design, CISC instruction decoding, variable-length ISA, branch predictor, code caching.

1 Introduction

Several current microprocessors like the Intel Pentium family [1] implement CISC instruction set architectures. Processing these CISC instructions requires higher design complexity than processing simple fixed-size RISC instructions. A widespread strategy to deal with CISC instructions is to decode them into simple RISC microoperations, which can be efficiently managed and executed by the processor back end. In this context, fast instruction fetch and decoding becomes critical for feeding the processor back end with enough instructions to keep the execution engine busy and thus achieve high performance.

However, it is not easy to design a fast decoding mechanism for CISC microprocessors. CISC instructions can have variable length, and a complex logic is required to decode instructions that can start at any byte address and can be translated into one or several microoperations. The decoding mechanism of the Intel P6 architecture is a clear example [2]. A complex instruction that produces multiple microoperations can only be decoded when it is the first instruction decoded in a cycle. This means that the decoding logic stalls when it finds a complex instruction that is not in the first decoding slot. The decoding process cannot continue until the next cycle, when the complex instruction reaches the first decoding slot after all the preceding instructions have been decoded. On the average, we have found that 18 percent of dynamic instructions are complex instructions in our benchmark programs. Consequently, this decoding strategy seriously limits the decoding speed. In other words, although the fetch architecture of a CISC processor provides high instruction fetch bandwidth, it could be not enough if decoding the fetched instructions becomes a bottleneck.

A well-known mechanism to overcome this problem is the trace cache [3], [4], [5]. The trace cache fetch architecture provides high fetch performance by buffering and reusing dynamic instruction traces. These traces are portions of the dynamic program execution that may contain multiple basic blocks, that is, several branches regardless of them being taken or not. Traces are dynamically built after their instructions have been decoded. Thus, the instructions stored in the trace cache are already decoded, which means that there is no need to decode the instructions fetched from it. As a result, the complexity of decoding instructions is removed from the critical path most of the time, since the instructions should only be decoded when there is a trace cache miss.

Fig. 1 shows the performance slowdown caused by the P6 decoding strategy in a processor implementing a trace cache fetch architecture similar to the one described in [6]. These performance results, measured in microoperations per cycle, are obtained using the superscalar processor model described in Section 5. Data are provided for 10 programs from the SPECint2000 benchmark suite, compiled using the x86 instruction set architecture, and for two different processor widths. The baseline processor

- *O.J. Santana is with the 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. Falcón is with the Barcelona Research Office, Hewlett-Packard Laboratories, Avda Graells, 501, Sant Cugat del Valles, 08174 Barcelona, Spain. E-mail: ayose.falcon@hp.com.*
- *A. Ramirez and M. Valero are with the Universitat Politècnica de Catalunya and with Barcelona Supercomputing Center, Campus Nord UPC, Jordi Girona, 1-3, 08034 Barcelona, Spain. E-mail: {aramirez, mateo}@ac.upc.edu.*

Manuscript received 15 June 2007; revised 22 Jan. 2008; accepted 28 July 2008; published online 8 Sept. 2008.
Recommended for acceptance by M. Gokhale.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2007-06-0239.
Digital Object Identifier no. 10.1109/TC.2008.170.

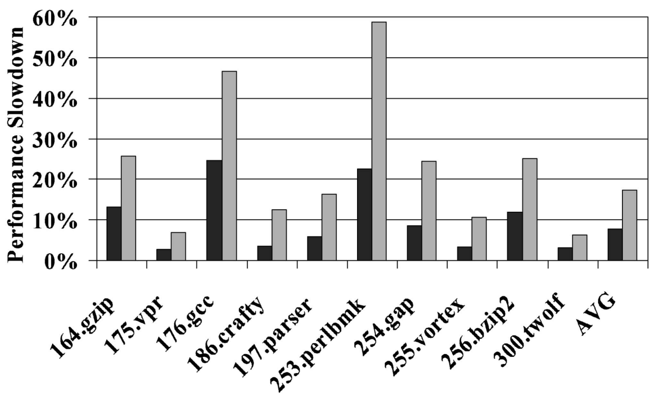


Fig. 1. Performance slowdown against ideal decoding of a superscalar processor implementing a P6-like decoding strategy and a trace cache assuming that the trace cache stores nondecoded instructions.

uses an ideal decoding mechanism able to decode as many instructions per cycle as the processor width.

Assuming that the trace cache stores nondecoded instructions, it is clear that the decoding bottleneck becomes a serious performance limiting factor due to the frequent appearance of instructions that should be decoded to multiple microoperations. A 4-instruction-wide processor suffers from an average of 8 percent performance degradation, which ranges from 3 percent in *175.vpr* to 25 percent in *176.gcc*. This bottleneck becomes more harmful for a wider processor, since it requires more instructions per cycle to keep its execution engine busy. An 8-instruction-wide processor suffers from an average of 17 percent performance degradation, which ranges from 6 percent in *300.twolf* to 59 percent in *253.perlbmk*.

However, if the trace cache is able to store already decoded instructions, the average performance slowdown is greatly reduced, being below 2 percent for all benchmarks in both processor setups. This means that the trace cache fetch architecture is an effective way for overcoming the instruction decoding bottleneck, but it is achieved at the cost of increasing the complexity of the fetch architecture. The trace cache needs more chip area and suffers from higher temperature and energy consumption than simpler fetch architectures organized around basic blocks. Fetching instruction traces requires not only a special-purpose storage—the trace cache—but also a secondary fetch mechanism for fetching instructions in case of a trace cache miss.

This paper proposes an alternative for exploiting the benefits of fetching already decoded instructions, while avoiding the increase in the fetch engine complexity caused by hardware code caching techniques like the trace cache. Our proposal is to store already decoded instructions in a special memory area allocated by the operating system for the program being executed. This memory area, namely, the Decoded Instruction Area (DIA), is managed using the branch prediction architecture. DIA contains blocks of already decoded instructions that correspond to the fetch blocks used as basic prediction units. When a new block of decoded instructions is introduced in DIA, the branch prediction mechanism is informed about the address where the decoded instructions are stored. Thus, when the branch prediction mechanism provides the address of a fetch block containing already decoded instructions, the fetch engine will be able to fetch decoded instructions from DIA instead of the original nondecoded instructions.

The operating system involvement lets our proposal take advantage from the hardware TLB translation and the operating system paging mechanism, just requiring to modify the operating system loader. In this sense, DIA is not like traditional code caching designs implemented in software. The main difference between DIA and other software code caching techniques such as Dynamo [7] and Code Morphing [8] is that the branch predictor is used to guide the mechanism. Consequently, DIA does not require any software overhead, since any code fragments are created beyond the basic prediction units. Moreover, these code fragments do not require to be rewritten in any way because they are naturally linked at runtime by the branch predictor itself.

Our decoding architecture can be implemented in conjunction with any branch prediction architecture. In this paper, we describe how to combine our proposal with the Fetch Target Buffer (FTB) branch prediction architecture [9], [10] and the stream fetch engine [11], [12]. Our results show that both the FTB-DIA and the Stream-DIA combinations are able to provide already decoded instructions most of the time, which allows our decoding architecture to achieve an important performance improvement over a processor implementing the P6 decoding strategy.

On the average, an 8-wide processor achieves 14 percent performance improvement when using either FTB-DIA or Stream-DIA. This improvement is comparable to the improvement achieved by a trace cache, but requiring lower implementation cost and complexity. In particular, Stream-DIA proves to be the most complexity-effective alternative. FTB-DIA requires 18 percent more area and 21 percent more energy consumption than Stream-DIA due to its more complex predictor structure, while the trace cache requires 40 percent more area and 36 percent more energy consumption than Stream-DIA due to its need for a secondary fetch engine to build traces. These results make Stream-DIA an excellent choice to balance cost and performance in front-end design for CISC processors with variable-length instructions.

2 The Decoded Instruction Area

Our proposal is based on storing already decoded instructions in the memory hierarchy. In order to do this, we use a fixed-size memory area called the DIA. DIA is allocated for the program being executed. When the operating system loads the program, it allocates DIA just like it allocates other segments. The DIA size is determined by the operating system loader for each particular machine implementation. The operating system communicates the DIA size to the processor by storing it in a special-purpose register.

Fig. 2 shows a simplified view of the structure of the memory allocated for a program using our decoding architecture. Like for the other segments, the loader assigns a number of pages in the logical address space for DIA. This means that DIA pages go through TLB translation, have the same operating system protection mechanisms, and can be swapped out just like any other memory page. Therefore, the operating system just requires slight modifications in the

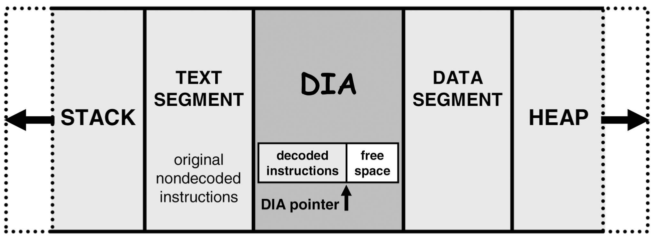


Fig. 2. Simplified view of the structure of the memory allocated by the operating system for a program using the DIA decoding architecture.

loader. It does not involve any compatibility problem with legacy codes because after updating the operating system, there is no need to modify the code of any application.

2.1 Interaction with the Branch Predictor

The storage of decoded instructions in DIA is guided by the branch prediction mechanism. Modern branch prediction architectures are organized to use sequences of instructions as basic prediction units [9], [11], [12], [13]. Each of these sequences of instructions constitutes a full fetch block finalized by a branch instruction. The starting address of a fetch block is used as index to access the branch prediction tables. Then, the branch predictor generates a prediction, which provides all the information required to fetch the full sequence of instructions and determine the starting address of the next fetch block.

Our decoding architecture takes advantage of the fact that the branch predictor is updated during the commit stage, when all the instructions belonging to a fetch block have committed. At this point, all these instructions are already decoded, and thus, our decoding architecture is able to store them in DIA. Therefore, the branch predictor is updated not only with the information required to predict the fetch block in the future but also with the memory address where a decoded version of the fetch block is stored. The next time this fetch block is predicted, the fetch architecture will search for the decoded version instead of the original nondecoded version, avoiding the need for decoding the instructions again.

The fetch blocks are stored in DIA following the order in which they are decoded. When a program starts execution, a pointer to the beginning of DIA is kept. As shown in Fig. 2, this pointer indicates the first free memory position of DIA where a decoded fetch block can be stored. When a new decoded fetch block is stored in the memory, the pointer is advanced to the end of the fetch block, that is, the new beginning of the free space.

The pointer never goes backward. DIA is flushed if the pointer reaches the end of the memory space assigned to DIA, that is, all the decoded fetch blocks stored in DIA are invalidated. Invalidating the fetch blocks stored in memory does not require modifying the memory contents. It is only necessary to return the DIA pointer to the beginning of DIA, as well as to invalidate the starting addresses of the decoded fetch blocks in the branch predictor. After that process, DIA is ready again to store new decoded instructions.

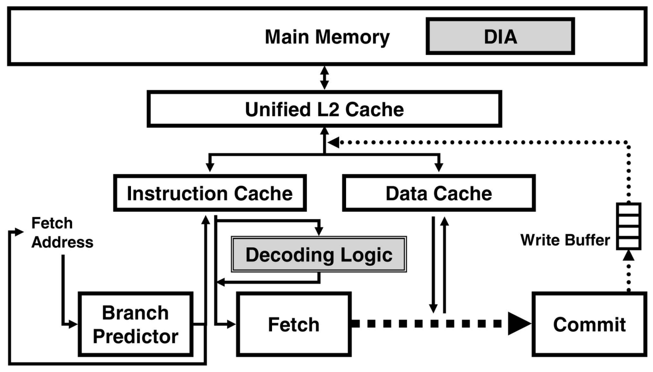


Fig. 3. The DIA decoding architecture.

2.2 Interaction with the Memory Hierarchy

Fig. 3 shows a block diagram of our decoding architecture. Like for the Pentium 4 processor [1], the pipeline has two paths: the fast path and the slow path. The fast path assumes that the instructions are already decoded, and thus, it does not use the CISC decoders, which are energy-intensive and a performance bottleneck. The slow path is the *emergency* path containing the complex CISC decoders. The instructions in the fast path must never arrive to rename before any older instruction that is still in the slow path.

Our cache model is virtually indexed, as well as physically tagged. Physical tags avoid the problem of synonym aliasing, while virtual indexes enable fast access to caches. Therefore, TLB accesses are not required to start cache accesses, but they are required to check the tags. This means that a TLB access is needed to update DIA contents during commit. Fortunately, DIA just needs a few pages in the logical memory space, and thus, the number of TLB conflicts does not suffer from a significant increase. Furthermore, a very small TLB could be included in the commit stage. This commit TLB would be a low-cost solution to avoid driving signals from the commit stage to the instruction TLB, which could be laid out far away in the chip.

According to the locality principle, new decoded fetch blocks must be kept near the processor to achieve high performance. Therefore, they are not directly written to main memory but to the second-level cache. In order to minimize off-chip memory traffic, our second-level unified instruction/data cache uses a write-back policy. Therefore, the decoded fetch blocks are stored in the main memory only when they are replaced from the second-level cache.

It is important to note that our proposal is absolutely transparent to the first-level instruction cache, and thus, no changes are required to the interface between this cache and the rest of the pipeline. It is also interesting to note that our second-level cache model has a single access port, which is shared by both the first-level instruction and data caches. This means that our decoding architecture does not need an extra access port. Every new decoded fetch block is introduced in a write buffer. The decoded fetch block will be stored in the second-level cache when the access port is free. Both instruction and data accesses are prioritized over storing decoded fetch blocks in the second-level cache.

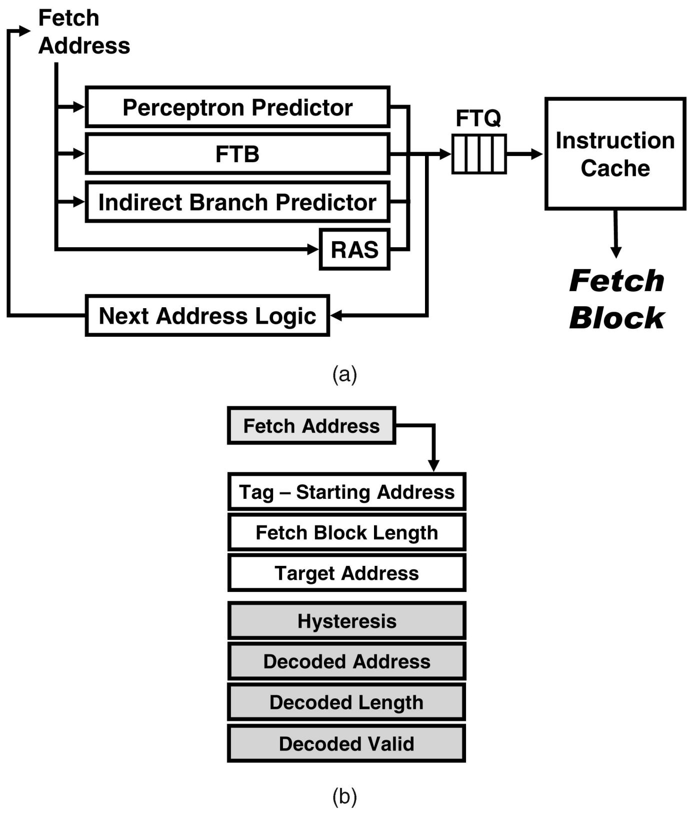


Fig. 4. The FTB branch prediction architecture. (a) Block diagram. (b) FTB structure.

2.3 Consistency of the Decoded Instructions

The decoded instructions stored in DIA must always be consistent with their associated original nondecoded version, but programs that modify themselves during execution change this relationship. Modern processors feature some kind of mechanism to invalidate instruction cache entries when a change in the code is detected. In the HP PA-RISC architecture [14], the program is expected to explicitly invalidate the instruction cache contents, forcing the cache to be refilled from memory. Other architectures like the Transmeta Crusoe [8] or the Intel Pentium 4 [1] feature some write-protecting mechanism of the memory pages being used by the programs. Self-modifying code is detected when a store tries to write in a protected page and then the cache contents are invalidated.

Our proposal can profit from any of these synchronizing techniques. Whatever the technique used, DIA is flushed when the instruction cache is invalidated. This strategy has the particular advantage that there is no need for detecting accesses to the memory pages assigned to DIA, which would be problematic because they are not write protected. More efficient techniques could be developed, but it is out of the scope of this work, since our benchmark programs do not modify their code during execution. Nevertheless, as flushing DIA is a very conservative model, it completely assures consistency.

3 Combining DIA with the Fetch Target Buffer

The FTB branch prediction architecture [9] is shown in Fig. 4a. This architecture constitutes a fully autonomous prediction engine capable of following a speculative path without further assistance. In each cycle, the branch predictor generates the fetch address for the next cycle and a fetch request that is stored in a Fetch Target Queue (FTQ). The instruction cache is then driven by the requests stored in the FTQ, effectively decoupling branch prediction from the memory access.

3.1 FTB Design

The FTB itself is a buffer that stores the information required to identify fetch blocks. A fetch block is composed by a sequence of instructions, stored in memory, which starts at a branch target and ends in a strongly biased taken branch. This mechanism allows strongly biased not taken branches to be embedded within a fetch block, increasing the fetch width without increasing implementation cost; as such, not taken branches can be easily predicted by simply ignoring them.

Given a fetch address, the FTB provides the length of the fetch block starting at that address, that is, the number of instructions belonging to the fetch block. The FTB also provides the type of the branch instruction finalizing the fetch block. If it is a conditional branch, then a conditional branch predictor is used to decide whether the branch is taken or not. Our model uses one of the most accurate state-of-the-art conditional branch predictors: The perceptron predictor [15]. If the branch finalizing the fetch block is a return instruction, a Return Address Stack (RAS) is used to obtain its target address [16]. Finally, if the branch instruction is an indirect branch, a special-purpose indirect branch predictor is used to obtain its target address [17]. All together, these four structures determine the destination of the branch finalizing the fetch block, which will be used as fetch address in the next cycle.

Combining DIA with the FTB branch prediction architecture is straightforward [10]. As described in Section 2, the FTB should keep not only the information required to provide a fetch block but also the address where the decoded version of the fetch block is stored. Storing the information of decoded fetch blocks requires adding new fields to the FTB, which are shown in Fig. 4b: The address where the decoded fetch block is stored, the decoded fetch block length (measured in bytes, since instructions may have different sizes), and a bit that indicates whether this information is valid or not. This bit is set to one when the data of a new decoded fetch block is introduced in the FTB. The valid bit is reset to zero if the fetch block is replaced from the FTB or after a DIA flush. We have checked using CACTI [18] that adding the new fields does not increase the number of cycles required to access the FTB and obtain a prediction. In addition, the increase in the branch predictor area is less than 30 percent, since the tag array is unmodified and no additional access port is required.

3.2 Decoded Fetch Block Selection

It is not necessary to store in DIA all the fetch blocks that appear during the execution of a program. Most program execution is concentrated in a reduced number of fetch blocks. In particular, we have found that 14 percent of the static fetch blocks that appear during the execution of our benchmark programs are responsible for 90 percent of the whole execution. Therefore, in order to efficiently use DIA, only those fetch blocks that are frequently executed should be stored.

To achieve this, we have added a hysteresis counter to each FTB entry, as shown in Fig. 4b. The hysteresis counter is used to decide whether a fetch block should be replaced from the FTB. When the predictor is updated with a new fetch block, the corresponding counter is increased if the new fetch block matches with the fetch block already stored in the selected entry. Otherwise, the counter is decreased, and if it reaches zero, the whole predictor entry is replaced with the new data, setting the counter to one. If the decreased counter does not reach zero, the new data are discarded.

A fetch block is stored in DIA only when the counter saturates, that is, when it reaches its maximum value. If the counter saturates, the decoded fetch block is stored in DIA, and the data required to access it are stored in the FTB, setting the valid bit to one. We have found that 4-bit hysteresis counters, increased and decreased by one, provide the best results. Therefore, a decoded fetch block is not introduced in DIA until it is executed at least 15 times. This number could be higher if a different fetch block tries to use the same table entry and decrements the hysteresis counter before it saturates.

If a decoded fetch block is replaced from the FTB, the address where its decoded version resides is lost, and it cannot be accessed again. However, it does not mean that the decoded fetch block is removed from DIA. The decoded fetch block remains in DIA and becomes garbage, since its memory space cannot be reused until DIA is flushed. In case the fetch block is decoded again, it should be stored a second time in DIA, using a new memory position, thus wasting memory space. Fortunately, this situation happens a negligible percentage of the time, since our FTB hit rate is usually more than 98 percent.

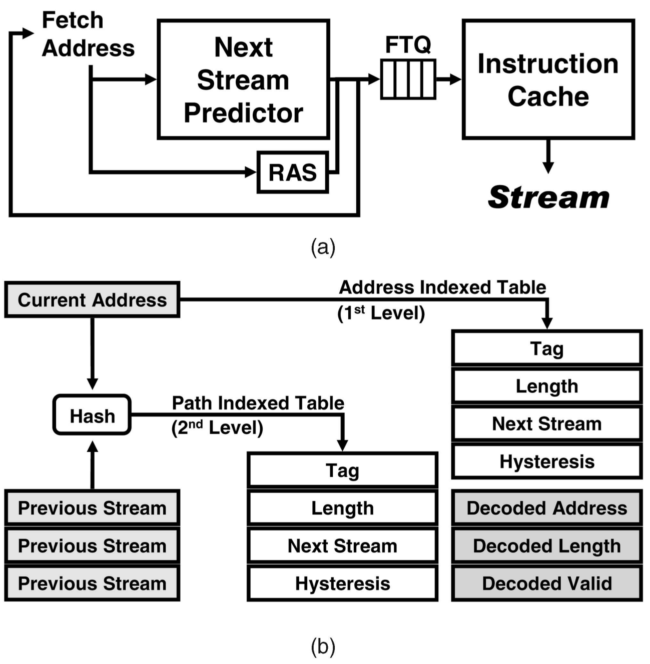


Fig. 5. The stream fetch engine. (a) Block diagram. (b) Cascaded stream predictor.

4 Combining DIA with the Stream Fetch Engine

The stream fetch engine [11], [12], which is shown in Fig. 5a, uses instruction streams as fetch blocks. A stream is a sequence of instructions, physically adjacent in memory, which starts at the target of a taken branch and finalizes at the next taken branch. The behavior of the branches contained inside the stream is implicit in the definition: All intermediate branches are not taken, and the last branch is taken.

The combination of our decoding architecture with the stream fetch engine will store decoded streams in DIA instead of decoded FTB fetch blocks. The main difference between an instruction stream and an FTB fetch block is that while FTB fetch blocks ignore biased not taken branches, the streams ignore all not taken branches. This means that instruction streams are longer than FTB fetch blocks, improving the fetch engine performance and, thus, the overall processor performance.

The core of this fetch engine is the next-stream predictor, a specialized branch predictor that provides stream-level sequencing. Given a fetch address, i.e., the current stream starting address, the stream predictor provides the current stream length, which indicates where the taken branch that finalizes the stream is. The predictor also provides the next stream starting address, which is used as the fetch address for the next cycle.

If the branch terminating the stream is a return instruction, a RAS [16] is used to predict its target address. However, unlike the FTB branch prediction architecture, the stream fetch engine does not require a separate conditional branch predictor or a separate indirect branch predictor, which simplifies the fetch engine design. The current stream starting address and the current stream length form a fetch request that is stored in an FTQ. The fetch requests stored in the FTQ are then used to drive the instruction cache.

4.1 Stream Predictor Design

Accurate stream prediction requires using path correlation with previously executed streams. To obtain an index into the prediction table, the fetch address and the contents of a history register with starting addresses of previous streams are hashed together using the same Depth-Older-Last-Current (DOLC) indexing scheme implemented in the trace predictor [19].

However, not all streams need path correlation to be accurately predicted. The next-stream predictor [11], [12] exploits this fact by implementing a cascaded structure [20], [21], which is shown in Fig. 5b. The prediction table is divided into two: A first-level table indexed only by the fetch address and a second-level table indexed using path correlation. A stream is introduced in the second level only if it is not accurately predicted by the first level. Therefore, those streams that do not need correlation are kept in the first level, preventing them from using correlation and thus avoiding unnecessary aliasing.

In order to generate a prediction, both levels are looked up in parallel. If there is a second-level table hit, its prediction is used. Otherwise, the prediction of the first-level table is used. The second-level prediction is prioritized because it is supposed to be more accurate than the first level due to the use of path correlation. However, the use of correlation also involves redundancy, that is, each stream is stored several times in the second-level table (one per each path followed to its starting address).

If a decoded stream is stored in DIA each time the second level is updated, the same decoded stream will be stored several times, wasting the available memory space. To avoid this, our stream decoding architecture only uses the first level for storing the information of decoded streams. The first level is indexed using just the fetch address, that is, the stream starting address. This involves that a stream is stored once and only once in the first-level table, avoiding unnecessary redundancy in DIA. As described for the FTB-DIA combination in Section 3, storing the information of decoded streams requires adding new fields to the first-level table: The decoded stream starting address, the decoded stream length, and a valid bit. We have also checked using CACTI [18] that these new fields cause less than 30 percent area increase and have no impact on the stream predictor access latency.

Therefore, only first-level predictions are able to provide the address where a decoded stream is stored with all its related information. If a stream is predicted by the first level, the decoded stream data can be directly used to fetch the stream. However, if a stream is predicted by the second level, the decoded stream data should be found in the corresponding first-level prediction. The length of both the first-level and the second-level predictions must be compared to assure that both levels predict the same sequence of instructions. If both levels predict the same stream length, the decoded stream data provided by the first level can be used despite the fact that the real prediction is done by the second level. Note that although the length provided by both predictions is the same, the target address of the two predicted streams could be different. Fortunately, more than 85 percent of second-level predictions coincide with the length provided by the first level, which makes this technique an efficient way of reducing redundancy in DIA.

4.2 Efficient DIA Usage

In order to efficiently use the available memory, DIA should keep only the frequently executed streams. As described in Section 3, our decoding architecture achieves this by using hysteresis counters. The original stream predictor design [11], [12] used hysteresis counters to decide if a stream should be replaced from the prediction table. Our decoding architecture uses these counters for a new purpose: A decoded stream is only stored in DIA when the corresponding first-level table hysteresis counter saturates. We have found that 4-bit hysteresis counters, increased and decreased by one, provide the best results.

However, there is an additional source of redundancy in this architecture. When a decoded stream is replaced from the first-level table, the address where its decoded version resides is lost. If the stream is decoded again in the future, it should be stored a second time in DIA, using a new memory position and thus wasting memory space. This is not a significant problem for the FTB branch predictor architecture because it does not store overlapping fetch blocks. If a taken branch is found halfway through a fetch block, the fetch block is split into two smaller parts. On the contrary, the stream fetch architecture allows for overlapping streams, choosing the appropriate one for each instance. Overlapping streams start at the same address so they should be kept in the same first-level table entry. Therefore, the probability of replacing a stream from the first-level table is higher than the probability of replacing a fetch block from the FTB.

Intuitively, decoded streams should not be commonly replaced. A stream is replaced because its 4-bit hysteresis counter reaches zero. However, it is not easy that the counter of a decoded stream reaches zero, not only because it has previously reached the maximum value, but also because the decoded stream is supposed to be frequently executed. This means that another frequently executed stream is trying to use the same table entry. Guided by this reasoning, we have measured how often each prediction table entry is replaced, finding that most replacements are concentrated in a low number of entries where two frequently executed streams collide.

Our stream decoding architecture takes advantage of this fact by using a small decoded stream victim cache [22]. When an already decoded stream is replaced from the first-level prediction table, it is stored in the victim cache. Each time the information of a new decoded stream is stored in the predictor, the victim cache is looked up. If the new stream is in the victim cache, it has already been decoded and stored in DIA. Therefore, the victim cache provides the memory address where the decoded stream is stored, removing the need for storing the decoded stream again. This also means that the victim cache should be flushed whenever DIA is flushed.

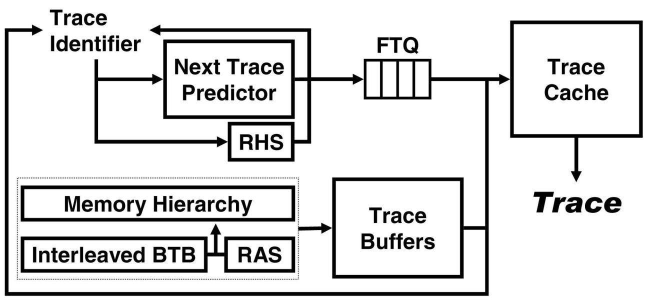


Fig. 6. The trace cache fetch architecture.

5 Experimental Methodology

The results presented in this paper have been obtained using a trace-driven simulation of a superscalar processor. Our simulator uses a static basic block dictionary to allow simulating the effects 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. Wrong-path instructions are never introduced in DIA, since they never commit.

5.1 The Trace Cache Fetch Architecture

Our simulator models the combination of our decoding architecture with the FTB branch prediction architecture [9], [10] described in Section 3 and the stream fetch engine [11], [12] described in Section 4. For comparison purposes, we also model a well-known mechanism for providing high fetch and decode bandwidth: The trace cache. We do not model a real trace cache design like the one used by the Intel Pentium 4 processor [1] because not all implementation details are public. Instead, we model the generic trace cache fetch architecture originally described in [6], which is shown in Fig. 6.

With the purpose of approximating the public details known about the Pentium 4 fetch architecture, we completely substitute the first-level instruction cache with a trace cache, and thus, trace cache misses are attended by the second-level cache. In order to assure a fair comparison with DIA, we have evaluated several trace cache setups, using and not using a separate instruction cache. We have found that the model presented here provides better results in terms of both overall performance and energy consumption.

In addition, we have enhanced this model by adding an FTQ [9] and hysteresis counters. Both the FTQ and the hysteresis counters behave like the corresponding elements in the DIA architecture: The FTQ decouples the next-trace predictor from the trace cache, while the hysteresis counters assure that only frequently executed traces are stored in the trace cache.

We faithfully implement the trace predictor described in [19], including alternate prediction. Trace predictions are stored in the FTQ, which feeds the trace cache with trace identifiers. An interleaved Branch Target Buffer (BTB) and a RAS [16] are used to build traces in the case of a trace cache miss. The BTB uses 2-bit saturating counters to predict the direction of conditional branches when a trace prediction is not available. This mechanism makes it possible to obtain instructions from the memory hierarchy and build new traces at a fast rate.

TABLE 1
Processors' Setup

fetch width rename/commit width int & fp issue width load/store issue width	4/8 instructions 4/8 instructions 4/8 instructions 2/4 instructions
int & fp issue queue load/store issue queue reorder buffer	32/64 entries 32/64 entries 128/256 entries
integer & fp registers	96/160
fetch target queue	4 entries
FTB	2048 entry 4-way FTB 256-entry perceptron pred. 2048 entry 4-way indirect pred. 32-entry overriding pred.
next stream predictor	1st level: 1024 entry 4-way 2nd level: 4096 entry 4-way
next trace predictor	1st level: 2048 entry 4-way 2nd level: 4096 entry 4-way
predictor latency	3 cycles
RAS and RHS	32 entries
L1 instruction cache	64Kb, 2-way, 1 port, 3 cycle
trace cache maximum trace size	512 traces, 4-way 32 micro-op (10 branch)
L1 data cache	64Kb, 2-way, 2/4 port, 3 cycle
L2 unified cache	1Mb, 4-way, 1 port, 16 cycle
main memory latency	350 cycles
page size	8KB
TLB	64-entry instruction TLB 128-entry data TLB 8-entry commit TLB

All the evaluated prediction architectures use a RAS to predict the target address of return instructions. However, the trace cache fetch architecture only uses the RAS during the trace building process. Instead of using a RAS, the trace predictor manages return instructions using a Return History Stack (RHS) [19], which keeps the trace history before the corresponding function call. The trace predictor does not use a history of previous trace starting addresses but a history of previous trace identifiers, and thus, the RHS is more efficient for trace prediction than a RAS.

5.2 Simulator Setup

We simulate two processor setups, a 4-wide and an 8-wide superscalar processor, both having a 20-stage pipeline. All microoperations are supposed to be 4 bytes long when they are stored in DIA or in the trace cache. We assume that in order to drive the corresponding signals, decoding requires three stages no matter whether the processed instructions are already decoded or not. This strategy also assures that the instructions already decoded do not arrive to rename before any older instruction that must be decoded using the CISC hardware decoders. The main values of our simulation setup are shown in Table 1.

The first-level instruction cache has a single access port and 64-Kbyte hardware budget. The trace cache fetch architecture replaces the instruction cache with a 64-Kbyte trace cache. The trace predictor is indexed using the DOLC scheme described in [19]. The stream predictor and the separate indirect branch predictor needed by the FTB architecture are indexed using the same DOLC scheme. We have explored a wide range of setups for all the evaluated prediction structures and selected the best one found.

The prediction tables modeled have a realistic three-cycle access latency, which has been calculated using CACTI [18] for a 0.10-μm technology. The overriding prediction technique [23] is used for tolerating the FTB access latency. The stream and trace predictors do not need an overriding predictor due to the long size of traces [24]. In addition, all predictors are decoupled from the corresponding instruction fetch mechanism using a four-entry FTQ. We have found that a larger FTQ does not provide additional performance improvements for the evaluated fetch models.

5.3 Benchmark Programs

We simulate 10 SPEC 2000 integer benchmarks.[1] Although we were not able to include data for programs with larger footprints, using SPECint2000 is not necessarily the best scenario for our proposal. Larger footprints will stress DIA more than integer programs, but they will also stress the trace cache. The advantage of DIA is that its size could be adjusted to find the optimal value for a particular program type, while the trace cache size is fixed by hardware design. Thus, larger footprints would highlight that DIA is still able to provide similar performance to the trace cache, as we show in Section 6 for the benchmark *186.crafty*.

We have compiled our benchmarks using the gcc 3.3.2 compiler with -O2 optimization level in an Intel Pentium 4 server under Red Hat Linux 7.1. A better compiler using code layout optimizations would have provided a higher quality code. However, as shown in [12], this kind of optimizations is less beneficial for the trace cache than for the FTB and stream fetch architectures, since the trace cache dynamically lays out the code together.

The x86 traces were collected from these benchmarks using the PIN instrumentation tool [25]. These traces contain 300 million x86 instructions obtained by executing the *reference* input set. We have analyzed the distribution of basic blocks as described in [26] in order to find the most representative execution segment for each benchmark. Finally, since the actual x86 microoperation model is not available for us, we translate the x86 instructions into microoperations using a decoding scheme based on the model provided by the rePLay transmogrifier tool [27]. The transmogrifier model leads to a scenario where just 18 percent dynamic instructions generate multiple microoperations, ranging from 11 percent in *176.gcc* to 24 percent in *253.perlbmk*.

6 DIA Evaluation

In this section, we evaluate the DIA decoding architecture. In order to understand DIA behavior, we explore the relationship of our architecture with both the memory hierarchy and the branch predictor. Then, we evaluate the performance of our proposal, as well as its efficiency in terms of chip area and energy consumption.

1. We do not simulate the benchmark *252.eon* because we have been unable to instrument it. In addition, we excluded *181.mcf* because its performance is very limited by data cache misses, being insensitive to changes in the fetch and/or decoding architecture.

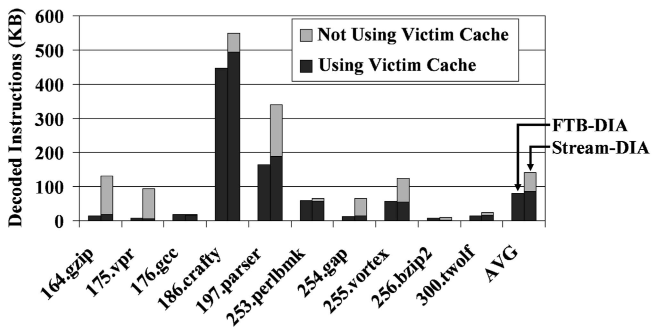


Fig. 7. Amount of memory (kilobytes) required to store all the dynamically decoded fetch blocks. Both the 4-wide and the 8-wide setup have similar behavior.

6.1 Memory Space Used

The total amount of memory space used by DIA determines its potential. Fig. 7 shows the memory space required to store all the instructions dynamically decoded by FTB-DIA and Stream-DIA. Stream-DIA needs slightly more memory space than FTB-DIA to store the decoded instructions due to redundancy. The stream fetch engine allows overlapping streams, and thus, it is more likely for Stream-DIA to store the same decoded instructions several times. The hysteresis counters make it possible to alleviate this problem, guaranteeing that only frequently executed instructions are stored in DIA. Indeed, if the hysteresis counters are not used, the memory space required would be six times higher due to the waste caused by storing infrequently executed instructions.

Fig. 7 distinguishes between the memory required whether or not Stream-DIA uses a decoded stream victim cache (FTB-DIA does not need a victim cache). The victim cache is an eight-entry fully associative cache having just 320 bits; we have found that it is enough for avoiding most situations in which two frequently executed streams try to use a single prediction table entry. On the average, the victim cache provides 40 percent reduction in the amount of memory required by Stream-DIA. Due to this reduction, the streams decoded by the Stream-DIA architecture require an amount of memory very similar to the one required by FTB-DIA. Using a victim cache has a negligible impact on the amount of memory required by FTB-DIA because the FTB does not allow overlapping fetch blocks.

We have evaluated a wide range of DIA sizes, and we have found that 64-Kbyte DIA achieves the best performance for both FTB-DIA and Stream-DIA. Most benchmarks require less than 64 Kbytes to store all their decoded instructions. The memory space required by benchmarks *175.vpr* and *256.bzip2* is even less than 10 Kbytes. Only the benchmarks *186.crafty* and *197.parser* require more than 64 Kbytes, forcing to occasionally flush DIA. Nevertheless, in spite of the high amount of memory space required, DIA flushes are not common. The benchmark *186.crafty* just flushes DIA every 38 million executed instructions in Stream-DIA and every 42 million executed instructions in FTB-DIA, while the benchmark *197.parser* flushes it every 100 million executed instructions in Stream-DIA and every 120 million executed instructions in FTB-DIA.

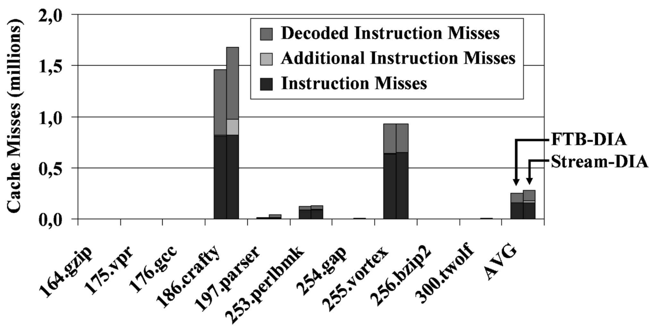


Fig. 8. Instruction cache misses. Both the 4-wide and the 8-wide setup have similar behavior.

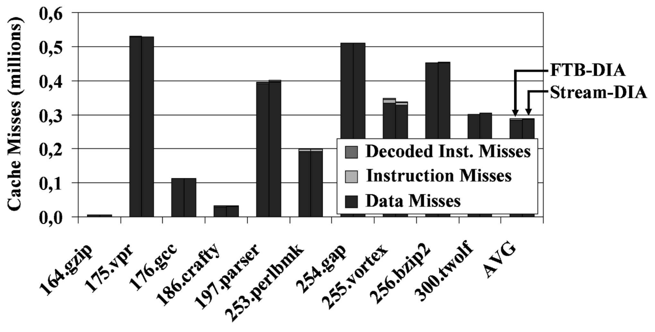


Fig. 9. Second-level cache misses. Both the 4-wide and the 8-wide setup have similar behavior.

6.2 Impact on the Memory Hierarchy

Storing decoded instructions in DIA avoids the need for decoding them again the next time they should be fetched. The first time a decoded fetch block is requested by the fetch engine, there should not be a compulsory miss in the second-level cache because new decoded fetch blocks are always introduced in the second-level cache. However, this first access causes a compulsory miss in the first-level instruction cache, which limits the achievable benefit.

Fig. 8 shows the total number of misses in the instruction cache (measured in millions). The bars are divided according to the cause of each miss. The lower part of each bar shows instruction cache misses caused by the original nondecoded instructions. These misses would also happen in a similar processor not using DIA. The middle part of each bar shows the additional instruction misses caused by conflicts with the decoded instructions. The higher part of each bar shows decoded instruction misses, that is, cache misses caused by fetching already decoded instructions from DIA.

As expected, storing decoded instructions in memory involves an increase in the total number of instruction cache misses. This increase is closely tied to the amount of additional information introduced in the memory. Stream-DIA suffers from more instruction cache misses because it requires more memory space to store the decoded instructions. However, this higher number of instruction cache misses is just relevant for the benchmarks *186.crafty* and *197.parser*, which are the ones that require more memory space.

The increase in the number of instruction cache misses is especially high for the benchmark *186.crafty*. This benchmark flushes DIA several times. Every DIA flush forces our architecture to start again the process of decoding instructions and storing them in memory, which causes more instruction cache misses. There is also a high increase for the benchmark *255.vortex* due to the fact that the original nondecoded instructions already cause a high number of cache misses when DIA is not used. On the contrary, the benchmark *197.parser* suffers from a relatively low number of cache misses. This benchmark requires a high amount of memory to store all the decoded instructions, but there is just a small subset of them that are frequently executed, thus limiting the amount of cache misses caused.

On the average, more than 90 percent additional cache misses are compulsory, that is, they are not due to conflicts in the cache, but to the fact that the decoded instructions have never been introduced in the instruction cache before. The benchmark *186.crafty* is the only one that suffers from a significant amount of additional misses due to conflicts between the original nondecoded instructions and the decoded instructions.

The higher number of instruction cache misses has little impact on the dynamic energy consumption of the instruction cache. Although the number of instruction cache misses is higher using DIA, it is still much lower than the total number of instruction cache accesses. We have measured, using CACTI [18], that the average increase in the instruction cache energy consumption is less than 2 percent for 0.10-μm technology. This slight increase in energy consumption is compensated by the overall reduction in the fetch architecture consumption due to our simpler design, as we show in the next sections.

Moreover, the impact of our technique on the second-level cache is minimal, in terms of both cache misses and dynamic energy consumption. Fig. 9 shows the total number of second-level cache misses (measured in millions). The bars are divided according to the cause of each miss: The program data, the decoded instructions, and the original nondecoded instructions. Data misses are by far the most important cause of second-level cache misses. There is just a slight increase in the number of second-level cache misses due to DIA. These misses should not be compulsory, since every new decoded fetch block is introduced in the second-level cache. Therefore, the additional misses are mostly caused by conflicts with the data or original nondecoded instructions stored in the second-level cache. Nevertheless, this increase in the number of second-level cache misses is absolutely negligible when compared with the number of misses caused by data. Furthermore, data are the most important source of write backs in the second-level cache, and thus, the additional write backs generated by DIA have no significant impact.

6.3 Decoded Instruction Coverage

Fig. 10 shows the decoded instruction coverage for FTB-DIA, Stream-DIA, and the trace cache. We call coverage to the percentage of correct-path executed instructions that were fetched already decoded. The main observation from this figure is that in spite of the increase in the number of instruction cache misses caused by our proposal, both FTB-DIA and Stream-DIA provides a high percentage of already decoded instructions. On the average, the three

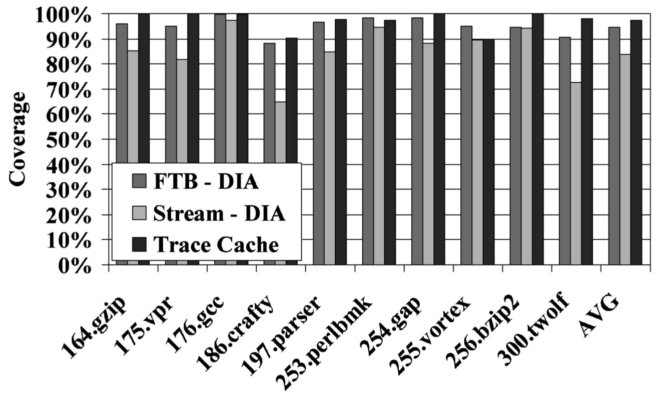


Fig. 10. Percentage of correct-path instructions that were fetched already decoded (coverage). Both the 4-wide and the 8-wide setup have similar behavior.

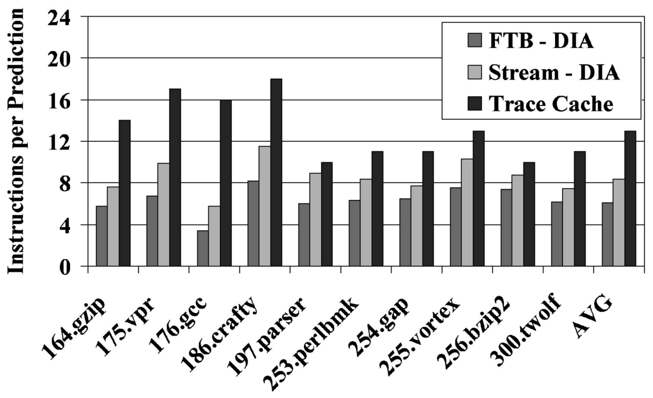


Fig. 11. Prediction unit length. Both the 4-wide and the 8-wide setup have similar behavior.

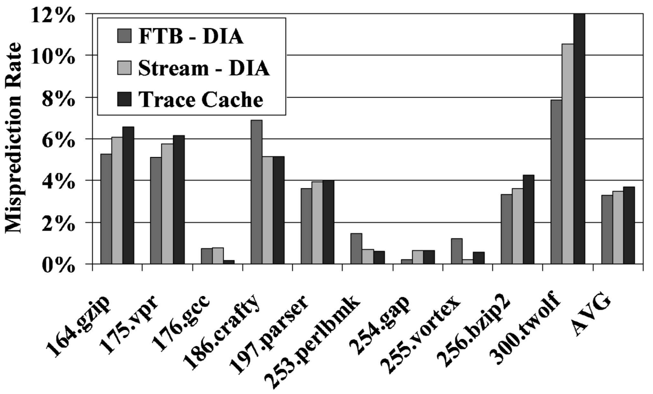


Fig. 12. Branch misprediction rate. Both the 4-wide and the 8-wide setup have similar behavior.

evaluated architectures provide already decoded instructions more than 80 percent of the time.

Stream-DIA coverage is lower than FTB-DIA coverage because redundancy is lower in the FTB. Although it would seem that the presence of overlapping streams in the stream predictor is a disadvantage, it is just this redundancy that allows the stream fetch engine to provide higher fetch bandwidth than the FTB fetch architecture [11], [12]. The hysteresis counters and the decoded stream victim cache alleviate DIA redundancy for Stream-DIA, but there is still room for improvement.

It is interesting to note that FTB-DIA provides a coverage close to the trace cache for the benchmark *186.crafty*. Although there is a great amount of instructions that should be decoded for this benchmark, they are more problematic for the trace cache than for DIA because the total number and size of traces in the trace cache is limited by the hardware implementation. Since the maximum trace size is fixed, part of the available space in the trace cache is wasted due to traces that are shorter than the maximum size, while all the DIA space can be exploited to store decoded instructions. These data make us think that FTB-DIA would still provide close coverage to the trace cache when executing benchmarks with larger workloads.

6.4 Branch Predictor Behavior

DIA depends not only on the memory hierarchy but also on the branch predictor. Since DIA is guided by the branch predictor, the length of the basic prediction unit determines the length of the blocks of decoded instructions stored in DIA, and thus, it is an important factor of front-end performance.

Fig. 11 shows the average number of instructions per prediction for the evaluated fetch architectures. Instruction traces are the longer prediction unit. The higher number of instructions per prediction enables the trace cache architecture to achieve a higher fetch bandwidth. In other words, the trace cache is the fastest fetch architecture evaluated. Instruction streams are shorter than traces because they can only contain a single taken branch. However, streams are never broken by not taken branches, and thus, they are longer than FTB fetch blocks, making the stream fetch engine faster than the FTB architecture.

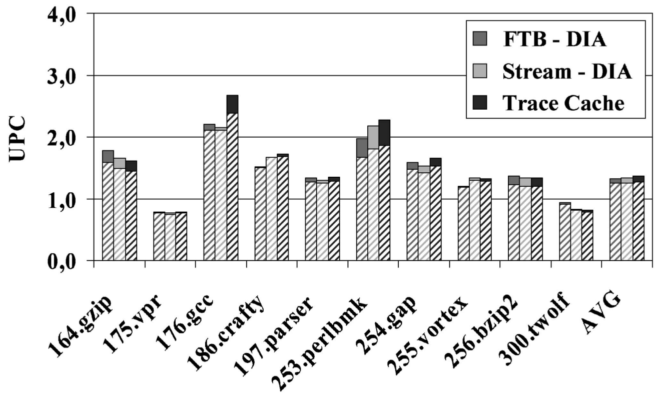
However, a fast fetch architecture is not enough to assure high performance. Providing a lot of instructions per cycle is useless if these instructions do not belong to the correct execution path. Therefore, branch prediction accuracy is another vital factor of front-end performance.

Fig. 12 shows the branch misprediction rate of the three evaluated branch predictors. On the average, all of them provide accurate predictions, since the average misprediction rate is below 4 percent. However, they are not equally accurate. The FTB provides the most accurate predictions due to the perceptron algorithm, while the trace predictor is the least accurate. This means that although the trace cache provides more instructions per cycle, there are a higher number of those instructions that are discarded due to branch mispredictions.

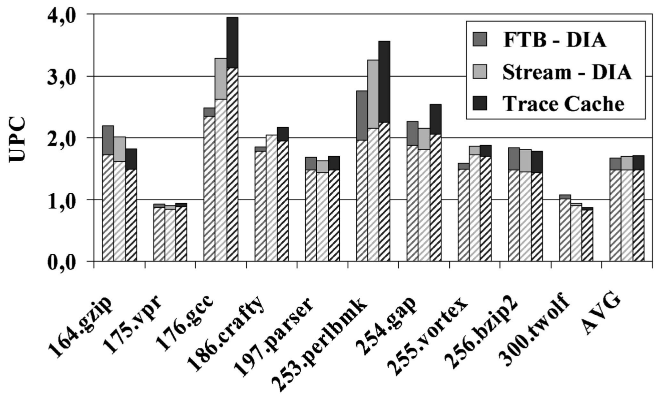
The general trend in the evaluated fetch architectures is that the more accurate the branch predictor is, the less instructions per prediction it provides. It cannot be considered a strict rule, since different fetch architectures would have different behaviors, but evaluating it is out of the scope of this work. The main observation from these data is that, as we show in the next section, the ability of fetching more instructions per cycle is compensated by a lower accuracy, leading to a scenario in which the three architectures provide similar performance.

6.5 Processor Performance

DIA fetches already decoded instructions from memory. This makes it possible to bypass the decoding logic, which



(a)



(b)

Fig. 13. Processor performance (microoperations per cycle). The shadowed part of each bar shows the performance achieved by any of the fetch architectures when the decoding capabilities are disabled. (a) 4-wide processor. (b) 8-wide processor.

improves performance and saves energy. However, the disadvantage of our technique is that it increases the total number of instruction cache misses, limiting the performance gain. Fig. 13 examines this trade-off. It shows the performance achieved by FTB-DIA and Stream-DIA, as well as the performance achieved by our trace cache model as the relative comparison point. Data are measured in microoperations per cycle (UPC) and provided for both the 4-wide and the 8-wide processor setup. In addition, the shadowed part of each bar shows the performance achieved by any of the fetch architectures when the decoding capabilities are disabled, i.e., when all fetched instructions must always be decoded.

FTB-DIA and Stream-DIA provide important performance improvements. The average improvement of adding the DIA decoding capabilities in the 4-wide processor setup is 5 percent for both DIA implementations. This improvement is close to the 7 percent achieved by the trace cache. The improvement provided by our decoding architecture is higher for the 8-wide processor setup. The bottleneck caused by decoding instructions is a more limiting factor for this wider processor, which requires a higher number of instructions to keep its execution engine busy. On the average, the improvement of adding decoding capabilities to the 8-wide processor is 14 percent for both FTB-DIA and Stream-DIA. Once again, this performance improvement is close to the 16 percent improvement that can be achieved using a trace cache. Furthermore, these improvements are very close to an ideal decoding mechanism (not shown in the figures). On the average, FTB-DIA, Stream-DIA, and the trace cache suffer from less than 2 percent slowdown against an ideal decoding mechanism able to decode as many instructions per cycle as the processor width.

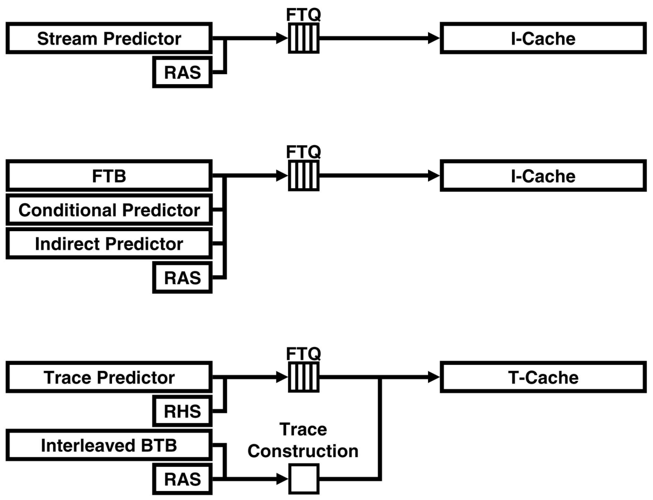


Fig. 14. Complexity comparison.

6.6 Fetch Engine Complexity

Overall, both FTB-DIA and Stream-DIA achieve performance close to our trace cache model: They are just around 2 percent slower than the trace cache. In addition, FTB-DIA and Stream-DIA achieve these performance results at a lower complexity than the trace cache. There are only two structures in the critical path of the fetch engines used by FTB-DIA and Stream-DIA: The branch predictor and the instruction cache. In these fetch architectures, decoded fetch blocks are mapped sequentially in memory and read from the instruction cache. On the contrary, instruction traces are not mapped sequentially in memory; they must be built and stored in a special-purpose hardware buffer. The need for a secondary fetch engine to build traces is exactly the main source of complexity of the trace cache architecture.

Fig. 14 shows a comparative diagram of the three evaluated fetch architectures. It becomes intuitively clear that Stream-DIA is the simpler implementation. The branch predictor and the instruction cache are the only structures in the critical path of the processor. In addition, there is just a branch prediction structure—the stream predictor—that works in conjunction with a RAS. FTB-DIA shares the same simple predictor-cache structure, but the branch predictor is more complex, since it requires separate tables for indirect and conditional branches.

The trace cache is the more complex implementation. The trace predictor (along with the RHS) and the trace cache have a predictor-cache structure that is similar to the stream predictor one. However, as mentioned above, a secondary path is required to build traces. The complexity of the interleaved BTB and the RAS is close to the trace predictor itself, and thus, the trace cache architecture requires almost twice the prediction resources used by Stream-DIA. Furthermore, additional resources are required to build instruction traces (trace buffers). Nevertheless, the higher complexity of the trace cache is alleviated by the lack of a separate first-level instruction cache. This solution has been adopted by real processor implementations [1].

We use CACTI [18] to measure the chip area required by the three evaluated fetch architectures as an approximation of their complexity. In order to increase the accuracy of the obtained measures, we have modified CACTI to model tagless structures and to work with setups expressed in bits instead of bytes. Our tool allows modeling all the structures required by the fetch engine of FTB-DIA, Stream-DIA, and the trace cache, including the prediction tables, the instruction and trace caches, the FTQ, etc. The full set of modeled structures is shown in Fig. 14.

Using 0.10-μm technology parameters, our results point out that Stream-DIA requires much less chip area than the other fetch architectures. FTB-DIA requires 18 percent more area than Stream-DIA due to its more complex predictor structure, while the trace cache requires 40 percent more area than Stream-DIA due to its need for a secondary fetch engine to build traces. Although interconnection wires between structures are not modeled, we consider that Stream-DIA would not require more area for them than FTB-DIA or the trace cache due to the relatively simpler bus layout.

6.7 Energy Consumption

Providing already decoded instructions makes it possible to save energy consumption, since the decoding logic is unused most of the time. We do not provide energy results of the CISC decoders because their layouts depend on particular machine implementations and not all details are public. Nevertheless, since FTB-DIA, Stream-DIA, and the trace cache provide already decoded instructions more than 80 percent of the time, it seems clear that using any of them will involve an important reduction in the energy consumption of the decoding hardware.

This reduction in the decoding consumption is not the only benefit of DIA. The lower complexity of the fetch architecture required by DIA also involves a reduction in the fetch engine consumption if compared with the trace cache. The higher trace cache consumption comes mainly from the need for a secondary 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 energy consumption compared to a system that always fetches instructions from the same location, like the FTB or the stream predictor. The latter has the particular advantage that it only uses a next-stream predictor, not needing the separate conditional and indirect branch predictors required by FTB-DIA. The next-stream predictor is not more complex that the trace predictor [19], which guarantees that Stream-DIA can achieve the same clock speed than the trace cache, if not higher.

In order to estimate the energy efficiency of the three fetch engines evaluated in this paper, we use our modified CACTI version to model the static energy consumption of all the structures required by the three evaluated architectures. These structures are shown in Fig. 14. Other structures such as the L2 cache, the write buffer, and the TLB suffer from minimal variation in energy consumption, and thus, they have no significant impact.

The static consumption results provided by CACTI represent, for each fetch engine structure, an approximation to the average consumption per access. After getting these results, we use our simulation tool to collect statistics about the number of dynamic lookup and update accesses to each structure. Both static consumption and dynamic accesses are then combined to obtain a measure of energy efficiency along program execution.

According to our estimations, Stream-DIA is the most energy-efficient architecture evaluated. FTB-DIA consumes 21 percent more energy than Stream-DIA due to its more complex predictor, while the trace cache consumes 36 percent more energy than Stream-DIA because a secondary fetch engine is required to build traces. Consequently, Stream-DIA proves to be an interesting complexity-effective architecture to implement CISC processors with variable-length ISAs due to its good performance results and its lower overall complexity.

7 Related Work

The trace cache fetch architecture is the result of a two-decade evolution. The fill unit [28] is one of the first attempts to dynamically collect already decoded instructions and store them in a special-purpose cache. A lot of research effort has been devoted to enhance the design of this special-purpose storage, leading to strategies like the decoded instruction cache [2], the microoperation cache [29], or the trace cache itself [3], [4], [5]. Finally, this evolution has made possible an actual physical implementation in the Intel Pentium 4 processor [1].

The trace cache does not eliminate the complex instruction decoder from the processor design, but it makes it possible to remove instruction decoding from the critical path, also allowing the decoder to be simplified. Our decoding architecture exploits the same idea: Already decoded instructions are fetched from DIA, allowing to bypass the decoder. Although removing the complex decoding logic from the critical path is not a new approach for the design of CISC microprocessors, we propose an innovative and straightforward implementation. The main advantage of our proposal is its simplicity, since it requires minimal hardware/software support. DIA uses hardware mechanisms already existing in current processor designs, not needing complex additional structures. Our proposal just requires adding some fields to the branch prediction tables, as well as to modify the L2 bus arbiter and include the DIA pointer, whose management logic is simple.

In general, DIA requires less hardware implementation cost than the trace cache. DIA does not need a special-purpose buffer to store the decoded instructions, since they are sequentially stored in the memory. As a consequence, DIA does not need a secondary fetch engine for fetching instructions in case of a miss in the special-purpose cache. This involves reducing the chip area and energy consumption, also avoiding problems with the chip temperature, since the trace cache is a well-known hot spot.

7.1 Software Code Caching

Techniques for code caching have not only been implemented in hardware. Dynamo [7] is a dynamic optimization system that is implemented entirely in software. Frequently executed instruction sequences are detected and stored in a fixed-size memory area. These instructions, namely, the hot code, are processed to create optimized sequences of instructions, called fragments. Fragments are stored in the memory by a linking mechanism, which also connects fragment exit branches to other fragments in the memory if possible.

DAISY [30] is a system designed to emulate existing architectures, making it possible that binaries generated for a particular architecture run on a simple VLIW core without requiring any modification. When a new part of the code is executed for the first time, it is translated to the VLIW instruction set and stored in a special part of the main memory by a virtual machine monitor. Caching the translated instructions in the memory prevents subsequent executions of the same part of the code from requiring translation again.

DELI [31] also caches a copy of the program being executed, but this is done for a different purpose. DELI makes it possible to observe and manipulate the instructions of the running program just before they are executed. In this way, DELI allows fine-grain control over the execution of programs by providing an interface to the layer between hardware and the execution of software, simplifying the design of client programs such as microarchitecture emulators.

The Transmeta Crusoe processor [8] uses a software layer, namely, Code Morphing, which is similar in spirit to DAISY. Code Morphing enables x86 instructions to be executed in a VLIW hardware core. The native instruction set of the VLIW core bears no resemblance to the x86 instruction set. However, Code Morphing translates the instructions and stores them in the memory, making it possible to reutilize them and, at the same time, enabling dynamic optimization.

BOA [32] is a second-generation DAISY architecture that is based on collecting and exploiting runtime system information in order to dynamically optimize code and reoptimize it for specific workload behavior. The dynamic runtime optimization system is executed on a simple architecture designed to achieve high frequency. In addition, the dynamic compilation system allows customizing the underlying execution engine and completely redefines the hardware interface, while maintaining binary compatibility at the software level.

It is interesting to note that all these systems suffer from software overhead. On the contrary, DIA takes advantage of hardware mechanisms already existing in the processor, not needing a software layer to manage its operation. In this sense, DIA is an interesting contribution to code caching systems because it could automatize critical tasks and reduce software overhead. Although evaluating particular alternatives is out of the scope of this paper, this would be an interesting research field for future work.

For instance, in the case of Dynamo, fragment management would be considerably simpler using DIA. Since our proposal is guided by the branch predictor, the instruction sequences are naturally linked by the program control flow. DIA relies on the inherent capability of the branch predictor and the instruction cache to detect instruction locality and keep the most frequently executed instructions.

An additional advantage of our proposal is that DIA is not allocated in a fixed architecture-specified address. DIA is allocated by the operating system for the program being executed. The operating system involvement makes it possible for our architecture to use the hardware TLB translation and benefit from the operating system paging mechanism, just requiring to modify the operating system loader. Therefore, our technique does not need a software layer to manage DIA. Combining DIA with Code Morphing or DAISY would not allow entirely removing the software layer, but it would allow simplifying it and reducing the overhead.

7.2 Dynamic Code Optimization

Dynamic code optimization is a common feature of all these techniques. Both Dynamo and Code Morphing can dynamically optimize the instruction sequences stored in the memory. The trace cache functionality can also be expanded to include dynamic code optimization [33] but without suffering from software overhead. The rePLay [34] architecture uses a front end derived from the trace cache to generate long traces, called frames, which are dynamically optimized. Frames are stored in a frame cache and treated as atomic regions, potentially increasing the aggressiveness of the optimizations. PARROT [35] is a more recent proposal that gradually optimizes instruction traces, using a selective approach to apply complex mechanisms only upon the most frequently executed traces. This allows not only to improve the processor performance but also to reduce the trace cache energy consumption.

In-pipeline dynamic optimizers [36], [37] do not require any additional special-purpose storage; they just need a table-based hardware optimizer. These techniques do not divide the program into traces or frames but do continuous optimization, considering the full program as a whole and thus improving the quality of the optimizations performed. However, in-pipeline optimizers are on the critical path of the processor. Although it is out of the scope of this paper, dynamically optimizing instruction sequences before storing them in DIA is an interesting research topic for future work. DIA could be used to combine the best of the two worlds: Optimizations are done out of the critical path, in the commit stage like in trace cache architectures, but without needing a hardware trace cache to store the optimized code. Continuous optimization techniques can also be applied to improve the quality of the optimized code stored in DIA.

8 Conclusions and Future Work

The decoding logic is a performance bottleneck not only for x86-based processors but also for other CISC processors and vector processors. This bottleneck will become more severe due to the current technology trend toward deeper pipelines and higher clock frequencies. Hardware code caching architectures like the trace cache [3], [4], [5] overcome this problem by storing and fetching already decoded instructions. This approach removes the decoding logic from the critical path, but it is achieved at the cost of increasing the processor front-end complexity.

In this paper, we propose to store the decoded instructions not in a special-purpose hardware buffer but in a fixed-size memory area allocated by the operating system, reducing the front-end design complexity. Our proposal takes advantage of hardware structures already existing in the processor, extending the branch prediction mechanism with the ability to provide the memory address where a decoded version of the predicted instructions is stored. This strategy enables the fetch engine to provide already decoded instructions from the memory hierarchy, overcoming the decoding bottleneck.

In addition, our proposal can be combined with any branch prediction mechanism. We describe how to combine our decoding architecture with the FTB branch prediction architecture [9], [10] and with the stream fetch engine [11], [12]. Both combinations provide an average of 14 percent performance improvement in an 8-wide processor, which is comparable to the improvement achieved by using the more complex trace cache, but requiring less chip area and less energy consumption. On the average, our results show that Stream-DIA is the best trade-off between performance and complexity.

Furthermore, this is only a first step in this research line. Although we focus on storing decoded instructions, our proposal enables a plethora of future possibilities. It is possible to apply dynamic optimizations before storing the instructions in the memory, in a similar way as done by rePLay with frames [34] but without needing a special-purpose frame cache. It is also possible to apply continuous optimization techniques [36], [37] out of the critical processor path. More elaborate possibilities can be designed, such as rescheduling instructions to increase the available instruction-level parallelism, remapping instructions to improve the performance of the fetch engine, and even translating instructions into a different instruction set architecture.

These alternatives can be implemented using our architecture in isolation or combining it with existing systems like DAISY [30], Dynamo [7], and Code Morphing [8]. Our technique can contribute to the design of such systems with a straightforward way of selecting frequently executed instructions and managing control transfers between them. All these possibilities, along with the relatively low implementation cost required, turn our proposal into a worthwhile complexity-effective front end architecture.

ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education of Spain under Contracts TIN2007-60625 and CSD2007-00050, the HiPEAC European Network of Excellence, the Barcelona Supercomputing Center, and an Intel fellowship. The authors would like to thank Adrián Cristal, Germán Rodríguez, and Jeroen Vermeulen for their valuable help during the development of this work, as well as Daniel Ortega, Paolo Faraboschi, Tanausú Ramírez, and Alex Pajuelo for their worthwhile comments on the manuscript.

REFERENCES

[1] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Caerman, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J.,* vol. 5, no. 1, 2001.

[2] M. Smotherman and M. Franklin, "Improving CISC Instruction Decoding Performance Using a Fill Unit," *Proc. 28th Int'l Symp. Microarchitecture (MICRO),* 1995.

[3] 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.

[4] E. Rotenberg, S. Benett, and J.E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," *Proc. 29th Int'l Symp. Microarchitecture (MICRO),* 1996.

[5] D.H. Friendly, S.J. Patel, and Y.N. Patt, "Alternative Fetch and Issue Techniques for the Trace Cache Mechanism," *Proc. 30th Int'l Symp. Microarchitecture (MICRO),* 1997.

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

[7] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI),* 2000.

[8] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," *Proc. First Int'l Symp. Code Generation and Optimization (CGO),* 2003.

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

[10] O.J. Santana, A. Falcón, A. Ramirez, and M. Valero, "Branch Predictor Guided Instruction Decoding," *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2006.

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

[12] 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.

[13] T.Y. Yeh and Y.N. Patt, "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," *Proc. 25th Int'l Symp. Microarchitecture (MICRO),* 1992.

[14] A. Kumar, "The HP PA-8000 RISC CPU: A High Performance Out-of-Order Processor," *Proc. IEEE Symp. High-Performance Chips (Hot Chips),* 1996.

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

[16] 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),* 1991.

[17] P.Y. Chang, E. Hao, and Y.N. Patt, "Target Prediction for Indirect Jumps," *Proc. 24th Int'l Symp. Computer Architecture (ISCA),* 1997.

[18] P. Shivakumar and N.P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power and Area Model," Technical Report Research Report 2001/2, Western Research Laboratory, 2001.

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

[20] K. Driesen and U. Hölzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," *Proc. 31st Int'l Symp. Microarchitecture (MICRO),* 1998.

[21] 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 (ISHPC),* 2002.

[22] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Computer Architecture (ISCA),* 1990.

[23] 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 (MICRO),* 2000.

[24] 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.

[25] R. Cohn, D. Connors, W.C. Hsu, C.K. Luk, T. Moseley, H. Patil, and V.J. Reddi, "Software Instrumentation as a Tool for Architecture and Compiler Research," *Tutorial at the 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2004.
[26] T. Sherwood, E. Perelman, and B. Calder, "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications," *Proc. 10th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT),* 2001.
[27] *Replay Transmogrifier,* http://www.crhc.uiuc.edu/acs/tools/rpt/, 2007.
[28] S.W. Melvin, M.C. Shebanow, and Y.N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proc. 21st Int'l Symp. Microarchitecture (MICRO),* 1988.
[29] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen, "Micro-Operation Cache: A Power Aware Frontend for Variable Length Instruction Length ISA," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED),* 2001.
[30] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100 Percent Architectural Compatibility," *Proc. 24th Int'l Symp. Computer Architecture (ISCA),* 1997.
[31] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J.A. Fisher, "DELI: A New Run-Time Control Point," *Proc. 35th Int'l Symp. Microarchitecture (MICRO),* 2002.
[32] E. Altman and M. Gschwind, "BOA: A Second Generation DAISY Architecture," *Tutorial at the 31st Int'l Symp. Computer Architecture (ISCA),* 2004.
[33] D.H. Friendly, S.J. Patel, and Y.N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," *Proc. 31st Int'l Symp. Microarchitecture (MICRO),* 1998.
[34] S.J. Patel, T. Tung, S. Bose, and M.M. Crum, "Increasing the Size of Atomic Instruction Blocks Using Control Flow Assertions," *Proc. 33rd Int'l Symp. Microarchitecture (MICRO),* 2000.
[35] 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),* 2004.
[36] B. Fahs, T. Rafacz, S.J. Patel, and S.S. Lumetta, "Continuous Optimization," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA),* 2005.
[37] V. Petric, T. Sha, and A. Roth, "RENO—A Rename-Based Instruction Optimizer," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA),* 2005.

Oliverio J. Santana received the BSc and the MSc degree in computer science in 2000 from the Universidad de Las Palmas de Gran Canaria (ULPGC), Spain, and the PhD degree from the Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2005. He is an associated 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 has coauthored 25 international publications and is currently supervising two PhD students. He has served in the organization of several international conferences as an external reviewer, submission webmaster, and a member of the program committee. He is a member of the IEEE.

Ayose Falcón received the BSc and MSc degrees in computer science from the Universidad de Las Palmas de Gran Canaria in 1998 and 2000, respectively, and the PhD degree in computer science from the Universitat Politècnica de Catalunya (UPC) in 2005. His PhD research included fetch unit optimization, especially branch prediction and instruction prefetching. During his PhD years, he was a summer intern and then a consultant at Intel Microprocessor Research Laboratories and worked as a teach assistant at UPC for one year. Since 2004, he has been a research scientist at the Barcelona Research Office, Hewlett-Packard Laboratories, Barcelona. His current research interests include simulation and virtualization technologies, disciplines in which he has published several papers and disclosed six patents. He is an active member of the computer architecture community, serving as a reviewer of the most important conferences and being a member of the program committee of the Second Championship Branch Prediction Competition. He is a member of the IEEE.

Alex Ramirez received the BSc, MSc, and PhD (awarded the extraordinary award to the best PhD) degrees in computer science from the Universitat Politècnica de Catalunya in 1995, 1997, and 2002, respectively. He is an associated professor in the Computer Architecture Department, Universitat Politècnica de Catalunya, and a research manager in the Computer Architecture Group, Barcelona Supercomputing Center. He was a summer student intern with Compaq's WRL, Palo Alto, California, in 1999 and 2000 and with Intel's Microprocessor Research Laboratory, Santa Clara, in 2001. His research interests include compiler optimizations, high-performance fetch architectures, heterogeneous multicore architectures, and vector architectures. He has coauthored more than 80 papers in international conference proceedings and journals and supervised three PhD students.

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