

Branch Predictor Guided Instruction Decoding

Oliverio J. Santana
Universidad de Las Palmas
de Gran Canaria

ojsantana@dis.ulpgc.es

Ayose Falcón
Barcelona Research Office
HP Labs

ayose.falcon@hp.com

Alex Ramirez and Mateo Valero
Universitat Politècnica de Catalunya and
Barcelona Supercomputing Center

{aramirez,mateo}@ac.upc.edu

ABSTRACT

Fast instruction decoding is a challenge for the design of CISC microprocessors. A well-known solution to overcome this problem is using a trace cache. It stores and fetches already decoded instructions, avoiding the need for decoding them again. However, implementing a trace cache 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 buffer like the trace cache, 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 memory instead of the original non-decoded instructions. Our results show that an 8-wide superscalar processor achieves an average 14% performance improvement by using our decoding architecture. This improvement is comparable to the one achieved by using the more complex trace cache, while requiring 16% less chip area and 21% less energy consumption in the fetch architecture.

Categories and Subject Descriptors

C.1.1 [Processor Architectures]: Single Data Stream Architectures—*RISC/CISC, VLIW architectures*

General Terms

Design, Performance

Keywords

Instruction decoding, branch predictor, complexity-effective

1. INTRODUCTION

Several current microprocessors, like the Intel Pentium family [11], 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 micro-operations, 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 micro-operations. The decoding mechanism of the Intel P6 architecture is a clear example [29]. A complex instruction that produces multiple micro-operations 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 average, we have found that 18% dynamic instructions are complex instructions in our benchmark programs. Consequently, this decoding strategy seriously limits the decoding speed. Therefore, 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 [19, 23]. 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, so 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.

Figure 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 [24]. These performance results, measured in micro-operations per cycle, are obtained using the superscalar processor model described in Section 4. Data is provided for several programs from the SPECint2000 benchmark suite, compiled using the x86 instruction set architecture, and for two different processor widths. The baseline processor uses an ideal decoding

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

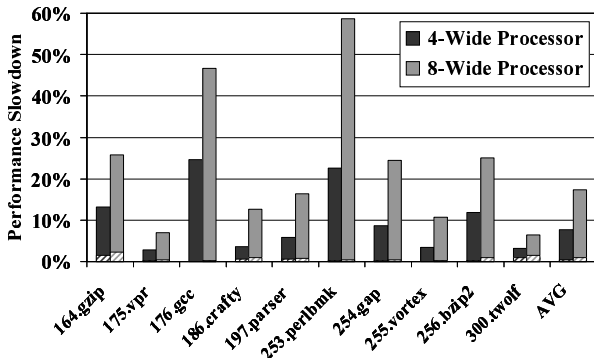


Figure 1: Performance slowdown against ideal decoding of a superscalar processor implementing a P6-like decoding strategy and a trace cache. Full bars assume that the trace cache stores non-decoded instructions; shadowed bars assume that the trace cache stores decoded instructions.

mechanism able to decode as many instructions per cycle as the processor width. Each full bar shows the slowdown caused by the P6 decoding strategy when the trace cache stores non-decoded instructions, while the shadowed part of each bar shows the performance slowdown when the trace cache stores already decoded instructions.

It is clear that the decoding bottleneck is a serious performance limiting factor due to the frequent appearance of instructions that should be decoded to multiple micro-operations. If the trace cache does not store already decoded instructions, a 4-instruction wide processor suffers from an average 8% performance degradation, which ranges from 3% in *175.vpr* to 25% 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 17% performance degradation, which ranges from 6% in *300.twolf* to 59% in *253.perlbmk*. Nevertheless, if the trace cache is able to store already decoded instructions, the performance slowdown is greatly reduced. This technique sets the average performance slowdown below 1% for both processor setups.

Therefore, the trace cache fetch architecture is an effective way for overcoming the instruction decoding bottleneck. However, this is obtained at the cost of increasing the complexity of the fetch architecture. The trace cache needs more area and suffers from higher temperature and energy consumption than simpler fetch architectures based on 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 a 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 non-decoded 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 [3] and Code Morphing [6], is that the branch predictor is used to guide the mechanism. Consequently, DIA does not require any software overhead, since no 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 linked at run time 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 [21]. Our results show that the FTB-DIA combination is 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 average, an 8-wide processor using our decoding architecture achieves 14% performance improvement. This improvement is comparable to the improvement achieved by a trace cache, but requiring lower implementation cost and complexity, since the fetch engine used by FTB-DIA requires 16% less chip area and it consumes 21% less energy.

The remainder of this paper is organized as follows. Section 2 presents our decoding architecture. Section 3 explains how to combine our decoding architecture with the FTB branch prediction architecture. Our experimental methodology is described in Section 4. Section 5 evaluates the FTB-DIA decoding architecture. We discuss previous related work in Section 6. Finally, Section 7 presents our concluding remarks and future research lines.

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 Decoded Instruction Area (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.

Figure 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,

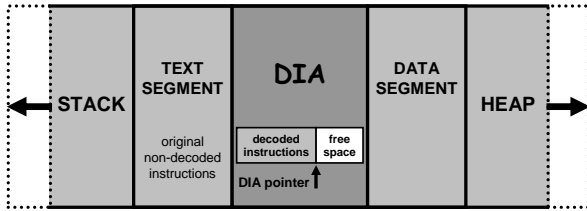


Figure 2: Simplified view of the structure of the memory allocated by the operating system for a program using the DIA decoding architecture.

the operating system just requires slight modifications in the 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 [31, 21]. Each one 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 non-decoded 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 Figure 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 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 to modify 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.

2.2 Interaction with the Memory Hierarchy

Figure 3 shows the block diagram of our decoding architecture. As for the Pentium 4 processor [11], the pipeline has

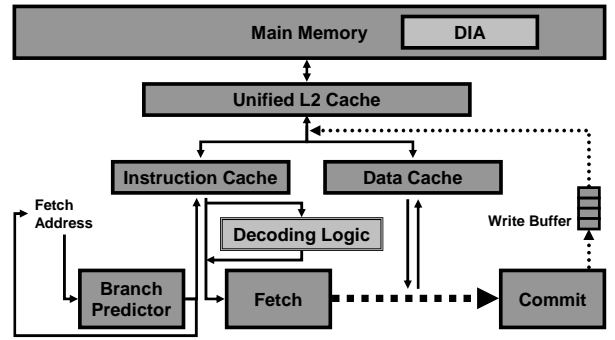


Figure 3: The DIA decoding architecture.

two paths: the fast path and the slow path. The fast path assumes that the instructions are already decoded, while the slow path is the *emergency* path containing the complex CISC decoders. The instructions in the fast path must not arrive to rename before any older instruction that is still in the slow path. Nevertheless, the fast path does not use the CISC decoders, which are energy-intensive and a performance bottleneck.

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.

It is interesting 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. The decoded fetch blocks are stored in the second level cache. Our second level unified instruction/data cache uses a write-back policy. According to this policy, the decoded fetch blocks are stored in memory only when they are replaced from the second level cache, minimizing the off-chip memory traffic.

It is also important to note that our second level cache model has a single access port, which is shared by both the instruction and data first level 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.

2.3 Consistency of the Decoded Instructions

The decoded instructions stored in DIA must always be consistent with their associated original non-decoded 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 [16], 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 [6] or the Intel Pentium [11], 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.

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 FTB

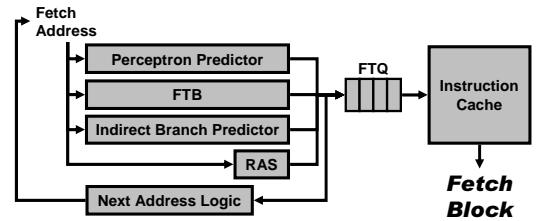
The Fetch Target Buffer (FTB) branch prediction architecture [21], which is shown in Figure 4.a, is composed of four elements. These four elements constitute a fully autonomous prediction engine, capable of following a speculative path without further assistance. Each cycle, the branch predictor generates 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

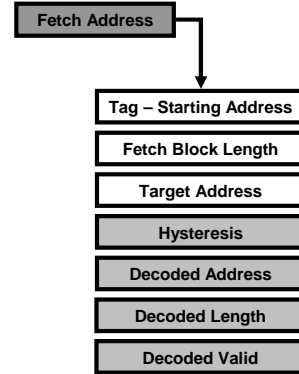
The FTB itself is a buffer that stores fetch blocks composed by a sequence of instructions starting at a branch target, and ending in a strongly biased taken branch. This mechanism allows strongly biased not taken branches to be embedded within a fetch block, increasing the fetch width without increasing 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 [14]. If the branch finalizing the fetch block is a return instruction, a Return Address Stack (RAS) is used to obtain its target address [15]. Finally, if the branch instruction is an indirect branch, a special-purpose indirect branch predictor is used to obtain its target address [4]. 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. 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 Figure 4.b: the address where the decoded fetch block is stored, the decoded fetch block length



(a) block diagram



(b) FTB structure

Figure 4: The FTB branch prediction architecture.

(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 [28] 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%, 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% of the static fetch blocks that appear during the execution of our benchmark programs are responsible for 90% 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 Figure 4.b. 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 is 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. 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, and thus wasting memory space. Fortunately, this situation happens a negligible percentage of the time, since our FTB hit rate is usually over 98%.

4. EXPERIMENTAL METHODOLOGY

The results presented in this paper have been obtained using trace driven simulation of a superscalar processor. Our simulator uses a static basic block dictionary to allow simulating the 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.

We simulate ten SPEC 2000 integer benchmarks¹. 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 type of programs, while the trace cache size is fixed by hardware design. Thus, larger footprints would highlight that DIA is still able to provide similar performance than the trace cache, as we show in Section 5 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 [25], this kind of optimizations are less beneficial for the trace cache than for the FTB architecture, since the trace cache dynamically lays out the code together.

The x86 traces were collected from these benchmarks using the PIN instrumentation tool [5]. These traces contain 300 million x86 instructions obtained executing the *reference* input set. We have analyzed the distribution of basic blocks as described in [27] in order to find the most representative execution segment for each benchmark. Finally, since the actual x86 micro-operation model is not available for us, we translate the x86 instructions into micro-operations using a decoding scheme based on the model provided by

¹We 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.

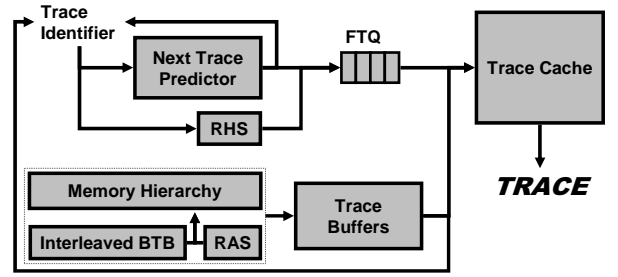


Figure 5: The trace cache fetch architecture.

the rePLAY transmogrifier tool [1], which leads to a scenario where just 18% dynamic instructions generate multiple micro-operations.

4.1 The Trace Cache Fetch Architecture

Our simulator models the combination of our decoding architecture with the FTB branch prediction architecture [21] described in Section 3. 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 [11], because not all implementation details are public. Instead, we model the generic trace cache fetch architecture originally described in [24], which is shown in Figure 5. In order to approximate 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 addition, we have enhanced this model by adding a Fetch Target Queue (FTQ) [21] to decouple the next trace predictor from the trace cache. We faithfully implemented the trace predictor described in [12], 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 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.

The FTB branch prediction architecture uses a RAS [15] 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), which keeps the trace history before the corresponding function call [12]. 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. All together, the trace prediction mechanism is able to provide branch prediction accuracy close to the FTB architecture using a perceptron predictor.

4.2 Simulator Setup

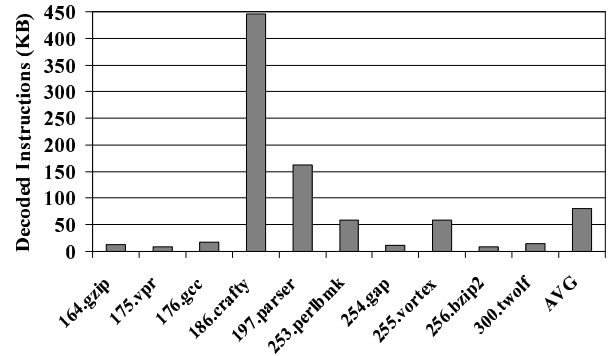
We simulate two processor setups, a 4-wide and an 8-wide superscalar processor, both having 20-stage pipeline. All micro-operations are supposed to be 4-byte long when they

Table 1: Processors setup.

fetch width	4/8 instructions
rename/commit width	4/8 instructions
int & fp issue width	4/8 instructions
load/store issue width	2/4 instructions
int & fp issue queue	32/64 entries
load/store issue queue	32/64 entries
reorder buffer	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 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	512 traces, 4-way
maximum trace size	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

are stored in DIA or in the trace cache. We assume that, in order to drive the corresponding signals, decoding requires 3 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 setups are shown in Table 1.

The first level instruction cache has a single access port and 64KB hardware budget. The trace cache fetch architecture replaces the instruction cache with a 64KB trace cache. The trace predictor, as well as the separate indirect branch predictor needed by the FTB architecture, are indexed using the Depth-Older-Last-Current (DOLC) scheme described in [12]. We have explored a wide range of setups for all the evaluated prediction structures, and selected the best one found. The prediction tables modelled have a realistic 3-cycle access latency, which has been calculated using CACTI [28] for a $0.10\mu\text{m}$ technology. The overriding prediction technique [13] is used for tolerating the FTB access latency. The trace predictor does not need an overriding predictor due to the long size of traces [26]. In addition, all predictors are decoupled from the corresponding instruction fetch mechanism using a 4-entry FTQ. We have found that a larger FTQ does not provide additional performance improvements for the evaluated fetch models.

**Figure 6: Amount of memory (kilobytes) required to store all the dynamically decoded fetch blocks.**

5. FTB-DIA EVALUATION

In this section, we evaluate the FTB-DIA decoding architecture. First of all, we explore the impact of our architecture on the memory hierarchy. Then, we measure and analyze the amount of already decoded instructions provided by our mechanism. Finally, we evaluate the performance of FTB-DIA and we compare it with the trace cache, also providing data about chip area and energy consumption.

5.1 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.

The increase in the number of instruction cache misses is closely tied to the amount of additional information introduced in memory. Figure 6 shows the total memory space required to store all the fetch blocks dynamically decoded by our mechanism (both 4-wide and 8-wide processor setups require similar memory space). We have evaluated a wide range of DIA sizes for FTB-DIA and we have found that 64KB DIA achieves the best performance. Most benchmarks require less than 64KB to store all their decoded fetch blocks. 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. The memory space required by benchmarks *175.vpr* and *256.bzip2* is even less than 10KB. Only the benchmarks *186.crafty* and *197.parser* require more than 64KB, forcing to occasionally flush DIA. Nevertheless, in spite of the high amount of memory space required, the benchmark *186.crafty* just flushes DIA every 42 million executed instructions, while the benchmark *197.parser* flushes it every 119 million executed instructions.

Figure 7 shows the total number of misses in the instruction cache (measured in millions). Data is shown for the 4-wide processor setup, since the 8-wide setup has similar behaviour. The bars are divided according to the cause of each miss. The lower part of each bar shows instruction misses, that is, instruction cache misses caused by the original non-decoded instructions. These misses would also hap-

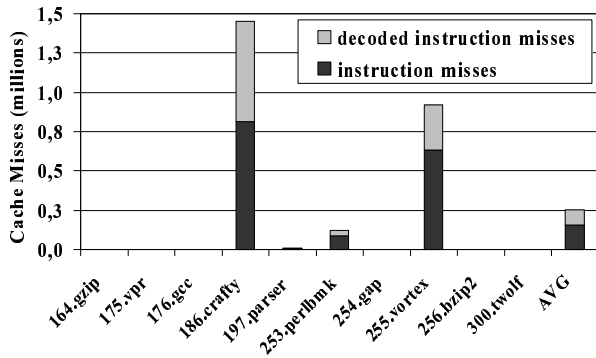


Figure 7: Instruction cache misses using FTB-DIA.

pen in a similar processor not using DIA, since we have found that the additional amount of instruction misses caused by conflicts with the decoded instructions is negligible when using FTB-DIA. 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 especially high for the benchmark *186.crafty*. This benchmark flushes DIA several times, forcing FTB-DIA 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 non-decoded 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. Although this benchmark requires a high amount of memory to store all the decoded instructions, there is just a small subset of them that are frequently executed, thus limiting the amount of cache misses caused.

The increase in the 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 remains relatively low when compared to the total number of instruction cache accesses. We have measured, using CACTI [28], that the average increase in the instruction cache energy consumption is less than 2% for a $0.10\mu\text{m}$ technology. This slight increase in energy consumption is compensated by the reduction in both static and dynamic energy consumption achievable due to our simpler design, as we show in the next subsections.

Moreover, the impact of our technique on the second level cache is minimal, both in terms of cache misses and dynamic energy consumption. 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. Furthermore, 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 are not compulsory, since every new decoded fetch block is introduced in the second level cache. Therefore, the additional misses are caused by conflicts with the data or original non-decoded 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.

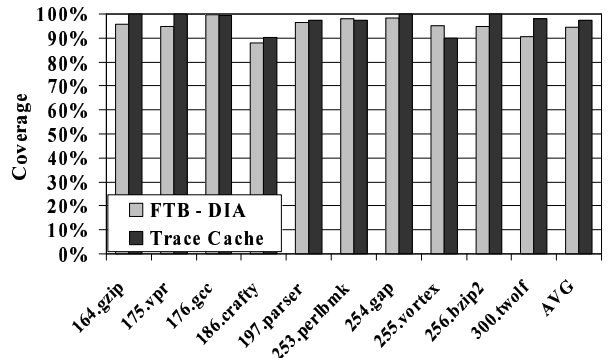


Figure 8: Percentage of correct-path instructions that were fetched already decoded (coverage).

5.2 Decoded Instruction Coverage

Figure 8 shows the decoded instruction coverage for FTB-DIA and the trace cache. Data are shown for the 4-wide processor setup because the 8-wide processor setup has similar behaviour. We call coverage to the percentage of correct-path executed instructions that were fetched already decoded. In spite of the increase in the number of instruction cache misses caused by our proposal, FTB-DIA provides a high percentage of already decoded instructions. Indeed, the percentage of already decoded instructions provided by FTB-DIA is close to the one provided by the trace cache.

The trace cache fetch architecture is faster, having a higher fetch bandwidth due to its ability of fetching beyond a taken branch during a single cycle. However, FTB-DIA has slightly better branch prediction accuracy due to the perceptron algorithm. Therefore, although the trace cache would supply a higher number of decoded instructions per cycle, there is also a higher number of those instructions that are squashed after branch mispredictions. Both factors compensate each other, allowing FTB-DIA to provide almost as much decoded instructions as the trace cache.

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.

5.3 Processor Performance

FTB-DIA fetches already decoded instructions from memory. This makes it possible to bypass the decoding logic, which 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. Figure 9 examines this trade-off. It shows the performance achieved by FTB-DIA, as well as the performance achieved by our trace cache model as relative comparison point. Data is measured in micro-operations per cycle (UPC) and provided for both the 4-wide and the 8-wide

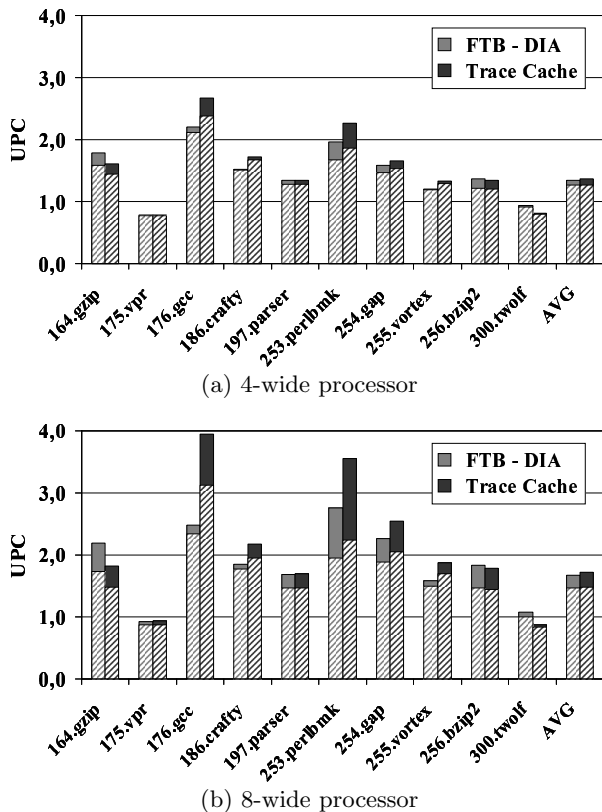


Figure 9: Processor performance for FTB-DIA and the trace cache (micro-operations per cycle).

processor setups. 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 provides important performance improvements. The average improvement of adding the DIA decoding capabilities to the FTB is 6% for the 4-wide processor setup, which is similar to the improvement achieved using a trace cache. The improvement provided by our decoding architectures 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 average, the improvement of adding decoding capabilities to the 8-wide processor is 14%. Once again, this performance improvement is similar to the 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 average, FTB-DIA and the trace cache suffer from less than 1% slowdown against an ideal decoding mechanism able to decode as many instructions per cycle as the processor width.

Overall, FTB-DIA achieves performance close to our trace cache model. On average, FTB-DIA provides performance only 2% lower than the trace cache, but at much lower complexity. Decoded fetch blocks are mapped sequentially in memory, and thus FTB-DIA does not require any redundant and/or complex storage like the trace cache, since it can use the instruction cache as the only source of decoded instructions. In addition, the need for a secondary fetch

engine is removed. The only structure that remains in the critical path is the branch prediction mechanism.

5.4 Chip Area and 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 both FTB-DIA and the trace cache provide already decoded instructions over 95% of the time on average, it seems clear that using any of them will involve an important reduction in the energy consumption of the decoding hardware.

We have used CACTI [28], configured with 0.10 μ m technology parameters, to model all the structures required by the fetch engine of both FTB-DIA and the trace cache. We modified CACTI to model tagless structures and to work with setups expressed in bits instead of bytes. This tool allows us to estimate the area and energy consumption of all fetch structures, including the prediction tables, the instruction and trace caches, the FTQ, etc. Other structures such as the L2-cache, the write buffer, and the TLB suffer from minimal variation in area and consumption, and thus they have no significant impact.

Although our tool is not able to model interconnection wires between these structures, we consider that they would require more area and energy for the trace cache model due to its higher complexity. Trace cache complexity comes mainly from the need for a second fetch engine to build traces. When there is a miss in the trace cache, instructions must be fetched, decoded, and packed into a trace from a secondary source. This second fetch engine increases cost, complexity, and area compared to a system that always fetches instructions from the same location, like the FTB. Overall, according to our estimations, the fetch engine of FTB-DIA requires 16% less chip area than the trace cache. Moreover, the fetch engine of FTB-DIA consumes 21% less energy than the trace cache. Since FTB-DIA is less complex and provides similar performance, it becomes an interesting complexity-effective alternative to the trace cache.

6. RELATED WORK

The trace cache fetch architecture is the result of a two-decade evolution. The fill unit [17] is one of the first attempts on dynamically collecting 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 [29], the micro-operation cache [30], or the trace cache itself [19, 23]. Finally, this evolution has made it possible an actual physical implementation in the Intel Pentium 4 processor [11]. Although the trace cache does not eliminate the complex instruction decoder from the processor design, it lets 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. We do not need a special-purpose buffer to store the decoded instructions, since they are sequentially stored in memory. As a consequence, we do not need a secondary fetch engine for fetching instructions in case of a miss in the special-purpose cache. This involves reducing chip area and energy consumption, also avoiding problems with the chip temperature, since the trace cache is a well-known hot spot.

Techniques for code caching have not only been implemented in hardware. Dynamo [3] 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 by an optimizing mechanism to create optimized sequences of instructions, called fragments. Fragments are stored in memory by a linking mechanism, which also connects fragment exit branches to other fragments in memory if possible. Dynamo could benefit from our proposal, since DIA management does not suffer from any software overhead. Our architecture is guided by the branch prediction mechanism, and thus the instruction sequences are naturally linked by the program control flow, relying on the inherent capability of the branch predictor and the instruction cache to detect instruction locality and keep the most frequently executed instructions.

The Transmeta Crusoe processor [6] uses a software layer, namely Code Morphing, for enabling x86 instructions to be executed in a VLIW hardware core. Code Morphing translates the instructions and stores them in memory, making it possible to reutilize them and, at the same time, enabling dynamic optimization. There are several proposals similar in spirit, like DAISY [8], DELI [7], and BOA [2], each one having its own particularities. The 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 to entirely removing the software layer, but it would allow to simplify it and reduce the overhead.

Dynamic code optimization is a common feature of all these techniques. Both Dynamo and Code Morphing can dynamically optimize the instruction sequences stored in memory. The trace cache functionality can also be expanded to include dynamic code optimization [10], but without suffering from software overhead. The rePLay [18] 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 [22] is a more recent proposal that gradually optimizes instruction traces, using a selective ap-

proach 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 [9, 20] 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 instructions 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 used to improve the quality of the optimized code stored in DIA.

7. CONCLUSIONS AND FUTURE WORK

Although instruction decoding is a well-known problem, this paper proposes a novel design that requires less implementation cost and complexity than previously proposed approaches. Our proposal takes advantage of hardware mechanisms already existing in the processor, not requiring special-purpose storage, like the trace cache, and yet it provides a competitive performance. Using the FTB branch prediction architecture, we have shown that DIA provides 14% performance improvement in an 8-wide processor, which is comparable to the improvement achieved by using the more complex trace cache, while requiring 16% less chip area and 21% less energy consumption in the fetch architecture.

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 memory, in a similar way as done by rePLay with frames [18], but without needing a special-purpose frame cache. It is also possible to apply continuous optimization techniques [9, 20] out of the critical processor path. In addition, the instructions can be rescheduled to increase the available instruction-level parallelism and remapped to improve the performance of the fetch engine. These alternatives can be implemented using our architecture in isolation or combining it with existing systems like DAISY [8], Dynamo [3], and Code Morphing [6]. 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.

8. ACKNOWLEDGEMENTS

This work has been supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01, the HiPEAC European Network of Excellence, the Barcelona Supercomputing Center, and an Intel fellowship. We would like to thank Adrián Cristal, Germán Rodríguez, and Jeroen Vermeulen for their help during the development of this work, as well as Daniel Ortega and Paolo Faraboschi for their worthwhile comments on the manuscript.

9. REFERENCES

- [1] Replay transmogripher.
<http://www.crhc.uiuc.edu/acs/tools/rpt/>.
- [2] E. Altman and M. Gschwind. BOA: A second generation DAISY architecture. *Tutorial at ISCA-31*, 2004.
- [3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *Conf. on Programming Language Design and Implementation*, 2000.
- [4] P. Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. *24th Intl. Symp. on Computer Architecture*, 1997.
- [5] 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 ASPLOS-XI*, 2004.
- [6] 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. *1st Intl. Symp. on Code Generation and Optimization*, 2003.
- [7] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. *35th Intl. Symp. on Microarchitecture*, 2002.
- [8] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. *24th Intl. Symp. on Computer Architecture*, 1997.
- [9] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. Continuous optimization. *32nd Intl. Symp. on Computer Architecture*, 2005.
- [10] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. *31st Intl. Symp. on Microarchitecture*, 1998.
- [11] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Caerman, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, 2001.
- [12] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. *30th Intl. Symp. on Microarchitecture*, 1997.
- [13] D. A. Jimenez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. *33rd Intl. Symp. on Microarchitecture*, 2000.
- [14] D. A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. *7th Intl. Conf. on High-Performance Computer Architecture*, 2001.
- [15] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th Intl. Symp. on Computer Architecture*, 1991.
- [16] A. Kumar. The HP PA-8000 RISC CPU: A high performance out-of-order processor. *Hot Chips VIII*, 1996.
- [17] S. W. Melvin, M. C. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. *21st Intl. Symp. on Microarchitecture*, 1988.
- [18] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. *33rd Intl. Symp. on Microarchitecture*, 2000.
- [19] 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*, 1995.
- [20] V. Petric, T. Sha, and A. Roth. RENO - A rename-based instruction optimizer. *32nd Intl. Symp. on Computer Architecture*, 2005.
- [21] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. *26th Intl. Symp. on Computer Architecture*, 1999.
- [22] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power awareness through selective dynamically optimized traces. *31st Intl. Symp. on Computer Architecture*, 2004.
- [23] E. Rotenberg, S. Benett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *29th Intl. Symp. on Microarchitecture*, 1996.
- [24] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48(2), 1999.
- [25] O. J. Santana, A. Ramirez, J. L. Larriba-Pey, and M. Valero. A low-complexity fetch architecture for high-performance superscalar processors. *ACM Transactions on Architecture and Code Optimization*, 1(2), 2004.
- [26] O. J. Santana, A. Ramirez, and M. Valero. Latency tolerant branch predictors. *Intl. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2003.
- [27] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *10th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2001.
- [28] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. *Western Research Laboratory, Research Report 2001/2*, 2001.
- [29] M. Smotherman and M. Franklin. Improving CISC instruction decoding performance using a fill unit. *28th Intl. Symp. on Microarchitecture*, 1995.
- [30] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen. Micro-operation cache: A power aware frontend for variable length instruction length ISA. *Intl. Symp. on Low Power Electronics and Design*, 2001.
- [31] T. Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *25th Intl. Symp. on Microarchitecture*, 1992.