Latency Tolerant Branch Predictors

Oliverio J. Santana, Alex Ramirez, and Mateo Valero, Fellow, IEEE
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona, Spain
email:{*osantana,aramirez,mateo*}@ac.upc.es

Abstract

The access latency of branch predictors is a well known problem of fetch engine design. Prediction overriding techniques are commonly accepted to overcome this problem. However, prediction overriding requires a complex recovery mechanism to discard the wrong speculative work based on overridden predictions.

In this paper, we show that stream and trace predictors, which use long basic prediction units, can tolerate access latency without needing overriding, thus reducing fetch engine complexity. We show that both the stream fetch engine and the trace cache architecture not using overriding outperform other efficient fetch engines, such as an EV8-like fetch architecture or the FTB fetch engine, even when they do use overriding.

1. Introduction

Fetch engine performance effectively limits the instruction level parallelism that can be exploited by wide-issue superscalar processors. This fact has led to the development of accurate branch prediction mechanisms, able to predict multiple branches per cycle. The EV8 [26] and the FTB [20] fetch architectures can predict multiple branches in a single cycle, whenever only the last one is taken. The next trace predictor [9] overcomes this limitation using traces as basic prediction unit. A trace is a fragment of the dynamic instruction flow, potentially containing multiple basic blocks, stored in a trace cache [16, 23].

The next stream predictor uses instruction streams [19] as basic prediction unit. We call stream to a sequence of instructions from the target of a taken branch to the next taken branch. Although a fetch engine based on streams is not able to fetch instructions beyond a taken branch in a single cycle, streams are long enough to provide a performance similar to a trace cache, outperforming both an EV8-like fetch architecture and the FTB fetch architecture. In addition, since streams are sequentially stored in the instruction cache, the stream fetch engine does not need a special-purpose storage, nor a complex dynamic building engine.

Nevertheless, besides providing high fetch bandwidth, fetch engine designers should take into account an additional problem: the branch predictor access latency. The continuous increase in processor clock frequency, as well as the larger wire latencies caused by modern technologies, prevent branch prediction tables from being accessed in a single cycle [1, 10]. In spite of this, the fetch address generation should be done in a single cycle to allow fetching instructions in the next cycle.

A common solution for this problem is using a small and fast predictor to obtain a first prediction in a single cycle. A slower but more accurate predictor provides a new prediction some cycles later, overriding the first prediction if they differ [10, 26]. Prediction overriding involves discarding the speculative work done based on the overridden prediction. However, this requires a complex mechanism, since those instructions fetched using the initial prediction should not be squashed if they will be fetched again using the new prediction [20].

In this paper, we analyze latency tolerant branch predictors, that is, prediction techniques able to provide a high performance without requiring and overriding mechanism. We focus on the length of the basic prediction unit. If predictions are long enough, the execution engine of the processor can be kept busy during multiple cycles, executing instructions from a long prediction, while a new prediction is being generated. Overlapping the execution of a prediction with the generation of the following prediction allows to tolerate the access delay of this second prediction, removing the need of an overriding mechanism, and thus reducing the fetch engine complexity.

We show that instruction streams and instruction traces are long enough to feed the processor back-end with instructions during multiple cycles, hiding the access latency of the next prediction. Therefore, both the stream and trace predictors are able to tolerate the access latency without needing a

Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'03)
1527-1366/03 $ 17.00 © 2003 IEEE

prediction overriding mechanism. Not using overriding reduces the performance of a stream fetch engine or a trace cache architecture, but they still outperform the FTB and EV8-like fetch architectures using overriding, requiring a lower complexity. This suggests an interesting tradeoff between reducing the fetch engine complexity and achieving a higher performance.

This paper is organized as follows. Section 2 exposes previous related work. Section 3 describes our simulation tool and benchmark suite, as well as the four evaluated fetch models. Section 4 shows that instruction streams and traces are longer than EV8 and FTB fetch blocks. Section 5 provides experimental results to backup our claims. Finally, section 6 exposes our concluding remarks.

2. Related Work

Superscalar processors require efficient fetch mechanisms, capable of providing multiple instructions per cycle, in order to keep their functional units busy. In addition, the fetch address generation should be done in a single cycle because this address is needed for fetching instructions in the next cycle. However, the increase in processor clock frequency, as well as the slower wires in modern technologies, cause branch prediction tables to require multi-cycle accesses [1, 10].

The trace predictor [9] is a latency tolerant mechanism, since each trace prediction is potentially a multiple branch prediction. A trace is a fragment of the dynamic control flow of the program, which is stored in a special-purpose trace cache [16, 23]. The processor front-end can use a single trace prediction to feed the processor back-end with instructions during multiple cycles, while the trace predictor is being accessed again to obtain a new prediction. Overlapping the prediction table access with the fetch of instructions from a previous prediction allows to hide the branch predictor access delay.

The next stream predictor [19] has the same ability, since a stream prediction is also a multiple branch prediction. A stream is a sequence of instructions starting in the target of a taken branch and finishing in the next taken branch, ignoring all intermediate not taken branches. The stream fetch engine requires a lower cost and complexity than the trace cache fetch architecture, since it does not need a special-purpose storage like the trace cache, nor a dynamic building mechanism. However, instruction streams are long enough to provide a performance similar to a trace cache.

The fetch target queue (FTQ) proposed in [20] is helpful for taking advantage of this fact. The FTQ decouples the branch prediction mechanism and the instruction cache access. Each cycle, the branch predictor generates the fetch address for the next cycle, and a fetch request which is stored in the FTQ. Since the instruction cache is driven by the requests stored in the FTQ, the fetch engine is less likely to stay idle while the predictor is being accessed again.

Another promising idea to tolerate the access latency is pipelining the branch predictor [13, 27]. Using a pipelined predictor, a new prediction can be started each cycle. Nevertheless, this is not trivial, since the result of a branch prediction is needed to start the next prediction. Therefore, a branch prediction can only use the information available in the cycle it starts, which has a negative impact on prediction accuracy. In-flight information could be taken into account when a prediction is generated, like described in [27], but this also involves an increase in the fetch engine complexity. It is possible to reduce this complexity in the fetch engine of a simultaneous multithreaded processor [30], as described by Falcon et al. [4], pipelining the branch predictor and interleaving prediction requests from different threads each cycle. However, analyzing the accuracy and performance of pipelined branch predictors is out of the scope of this paper.

Prediction Overriding

A different approach is the overriding mechanism described by Jimenez et al. [10]. This mechanism provides two predictions, a first prediction coming from a fast branch predictor, and a second prediction coming from a slower, but more accurate predictor. When a branch instruction is predicted, the first prediction is used while the second one is still being calculated. Once the second prediction is obtained, it overrides the first one if they differ, since the second predictor is considered to be the most accurate. A similar mechanism is used in the Alpha EV6 [6] and EV8 [26] processors, where a multi-cycle latency branch predictor overrides a faster but less accurate cache line predictor [2].

The problem of prediction overriding is that it requires an important increase in the fetch engine complexity. An overriding mechanism requires a fast branch predictor to obtain a prediction each cycle. This prediction should be stored for being compared with the main prediction. Some cycles later, when the main prediction is generated, the fetch engine should determine whether the first prediction is correct or not. If the first prediction is wrong, all the speculative work done based on it should be discarded. Therefore, the processor should track which instructions depend on each prediction done in order to allow the recovery process. This is the main source of complexity of the overriding technique.

Moreover, a wrong first prediction does not involve that all the instructions fetched based on it are wrong. Since both the first and the main predictions start in the same fetch address, they will partially coincide. Thus, the correct instructions based on the first prediction should not be squashed. This selective squash will increase the complexity of the re-

covery mechanism. To avoid this complexity, a full squash could be done when the first and the main predictions differ, that is, all instructions depending on the first prediction are squashed, even if they should be executed again according to the main prediction. However, a full squash will degrade the processor performance and does not remove all the complexity of the overriding mechanism.

Therefore, the challenge is to develope a technique able to achieve the same performance than an overriding mechanism, but avoiding its additional complexity. This paper is focused on using long basic prediction units to remove this complexity while still providing high performance. In particular, we focus on instruction streams and traces. The main difference of this paper from previous work on stream [19] and trace [9] predictors is the use of realistic access latencies to evaluate the performance achieved by these predictors. Our results show that the length of instruction streams and traces is enough to hide the branch predictor access latency without needing an overriding mechanism.

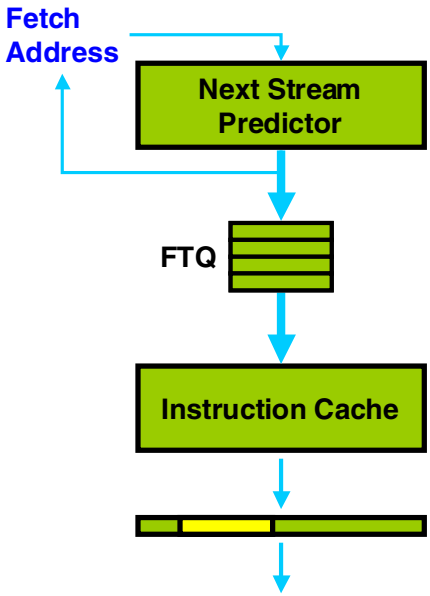


Figure 1. Stream fetch engine.

3. Experimental Methodology

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

We feed our simulator with traces of 300 million instructions collected from the SPEC 2000 integer benchmarks using the *ref* input set. To find the most representative execution segment we have analyzed the distribution of basic blocks as described in [28]. We excluded the benchmark *181.mcf* because its performance is very limited by data cache misses, being insensitive to changes in the fetch architecture. The average performance measures presented along the paper are the harmonic mean of the eleven benchmarks used.

Since previous work [18] has shown that code layout optimizations have a very important effect on all aspects of the fetch engine performance, we present data for both a baseline and an optimized code layout. The baseline code layout was generated using the Compaq C V5.8-015 compiler on Compaq UNIX V4.0, while the optimized code layout was later generated with the spike tool [3] shipped with Compaq Tru64 Unix 5.1. Optimized code generation is based on profile information collected by the pixie V5.2 tool using the *train* input set.

3.1. Fetch Models

We evaluate four state-of-the-art fetch architectures using prediction overriding: an EV8-like fetch architecture using a 2bcgskew branch predictor [26], the FTB fetch architecture [20] using a perceptron branch predictor [11], the stream fetch engine using a next stream predictor [19], and the trace cache architecture using a trace predictor [9].

The next stream predictor is evaluated using the stream fetch engine [19], shown in figure 1. The stream predictor access is decoupled from the instruction cache access using a fetch target queue (FTQ) [20]. The stream predictor generates requests which are stored in the FTQ. These requests are used to drive the instruction cache, obtain a line from it, and select which instructions from the line should be executed. In the same way, the remainder three fetch models use an FTQ to decouple the branch prediction stage from the fetch stage.

Our EV8-like fetch model is similar to the one described in [26] but modified to decouple the branch prediction mechanism from the instruction cache with an FTQ. An interleaved BTB is used to allow the prediction of multiple branches until a taken branch is predicted, or until an aligned 8-instruction block is completed. The branch prediction history is updated using a single bit for prediction block, which combines the outcome of the last branch in the block with path information, as described in [26]. Our FTB model is similar to the one described in [20] but using a perceptron branch predictor [11] to predict the direction of conditional branches. Figure 2 shows a diagram representing these two fetch architectures.

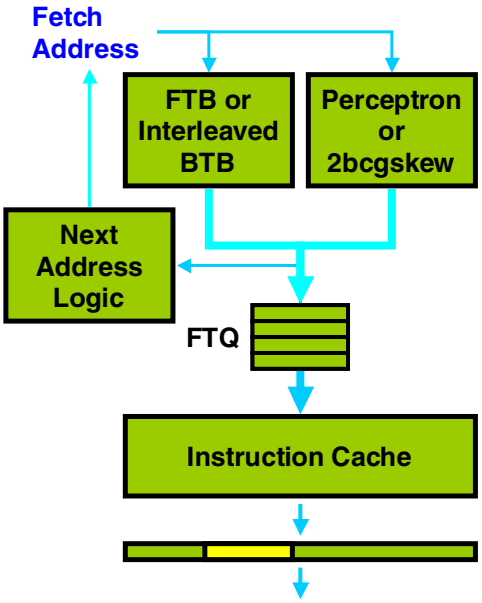


Figure 2. Fetch engine using an FTB or an interleaved BTB, as well as a separate conditional branch predictor (perceptron or 2bcgskew).

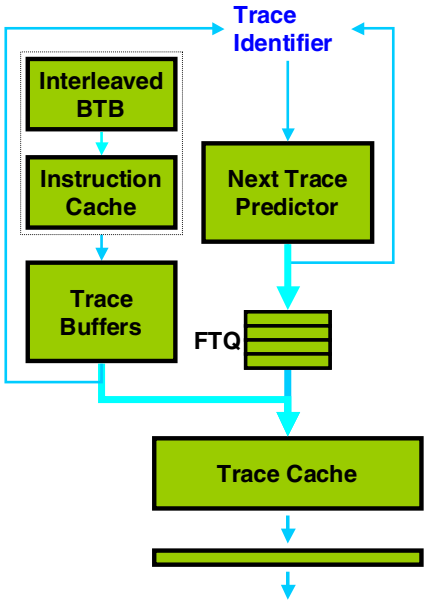


Figure 3. Fetch engine based on a trace predictor and a trace cache.

Our trace cache architecture model is similar to the one described in [24] but using an FTQ to decouple the trace predictor from the trace cache, as shown in figure 3. Trace predictions are stored in the FTQ, which feeds the trace cache with trace identifiers. An interleaved BTB is used to build traces in case of a trace cache miss. This BTB uses 2-bit saturating counters to predict the direction of conditional branches when a trace prediction is not available. In addition, an aggressive 2-way interleaved instruction cache is used to allow traces to be built as fast as possible. This mechanism is able to obtain up to a full cache line in a cycle, independent of PC alignment.

3.2. Processor Setup

The lower cycle time of current processors involves a larger number of stages in the processor pipeline. This fact has an impact not only on the branch predictor, but also on other processor components like the register bank or the memory hierarchy. Therefore, we have configured our simulator to model a 22-stage pipeline processor. We assume that 2 cycles are needed to access the register bank. We also assume 3 cycles to access the first level caches and 16 cycles to access the second level unified cache, according to data obtained using CACTI 3.0 [29] for a 0.10μm technology. Finally we assume that a main memory access requires 250 cycles.

In addition, we simulate two different processor setups: a 4-wide and an 8-wide superscalar processor. The fetch, issue, and commit widths of the second setup are twice those for the first setup. The wider setup also doubles the size of the issue queues and the reorder buffer, having a larger number of registers. These setups allow us to analyze the impact of the maximum fetch width in the performance of a multi-cycle branch predictor. The wider setup is able to execute a higher number of instructions each cycle, and thus it adds pressure to the branch prediction mechanism, since new predictions are required more frequently. The main values of these setups are shown in table 1.

Our instruction cache setup uses wide cache lines, that is, 4 times the processor fetch width [19], having a 64KB total hardware budget. The trace cache architecture is actually evaluated using a 32KB instruction cache, while the remainder 32KB are devoted to the trace cache. This hardware budget is equally divided into a filter trace cache [21] and a main trace cache. We use a longer maximum trace size in the 8-wide setup to help tolerating the branch predictor latency. Although longer traces reduce the total number of traces that can be kept in the trace cache, it provides a better performance in this setup than keeping a bigger number of shorter traces. In addition, we use selective trace storage [17] to avoid trace redundancy between the trace cache and the instruction cache.

4-wide processor	
fetch/rename/commit width	4 instructions
integer issue width	4 instructions
floating point issue width	4 instructions
load/store issue width	2 instructions
fetch target queue	4 entries
integer issue queue	32 entries
floating point issue queue	32 entries
load/store issue queue	32 entries
reorder buffer	128 entries
integer registers	96
floating point registers	96
register bank latency	2 cycles
filter/main trace cache	256 traces, 4-way
maximum trace size	16 inst. (6 branches)
8-wide processor	
fetch/rename/commit width	8 instructions
integer issue width	8 instructions
floating point issue width	8 instructions
load/store issue width	4 instructions
fetch target queue	4 entries
integer issue queue	64 entries
floating point issue queue	64 entries
load/store issue queue	64 entries
reorder buffer	256 entries
integer registers	160
floating point registers	160
register bank latency	2 cycles
filter/main trace cache	128 traces, 4-way
maximum trace size	32 inst. (10 branches)
memory hierarchy	
L1 instruction cache	64/32KB, 2-way
instruction cache block	4*fetch width bytes
L1 data cache	64KB, 2-way, 64B block
L1 latency	3 cycles
L2 unified cache	1 MB, 4-way, 128B block
L2 latency	16 cycles
main memory latency	250 cycles

Table 1. Simulator setup.

3.3. Prediction Tables Access Latency

We have measured the access time for the branch prediction structures evaluated in this paper using the CACTI 3.0 tool [29], a detailed wire and transistor structure model of cache memories. We modified CACTI to model tagless branch predictors, as well as to work with setups expressed in bits instead of bytes. Data we have obtained corresponds to a 0.10μm technology, which is expected to be used in a recent future.

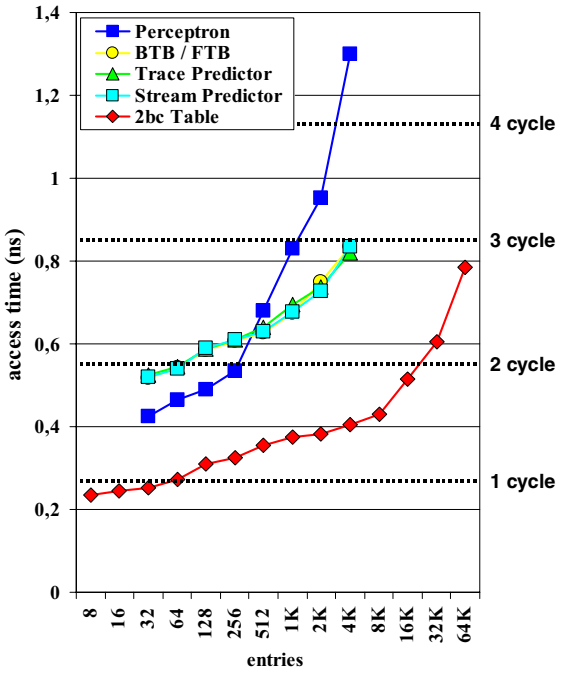


Figure 4. Access time to branch prediction tables in 0.10μm technology with a 3.47 GHz clock frequency.

For translating the access time from nanoseconds to cycles, we assumed an aggressive 8 fan-out-of-four delays clock period, that is, a 3.47 GHz clock frequency as reported in [1]. It has been claimed by Hrishikesh et al. [8] that 8 fan-out-of-four delays is the optimal clock period for integer benchmarks in a high performance processor implemented in 0.10μm technology.

Figure 4 shows the branch prediction table access time obtained using CACTI. We have measured the access time for a 2-bit counters table ranging from 8 to 64K entries. We have also measured the access time for a BTB, an FTB, a stream predictor, and a trace predictor ranging from 32 to 4K entry tables. These tables are assumed to be 4-way associative because direct mapped tables provide a poor performance, while 2-way associative tables require the same number of cycles to be accessed that 4-way associative tables in the evaluated setups.

Finally, we have measured the access time for a perceptron table ranging from 32 to 4K entries. Since the size of the table depends on the number of history bits, we have used 40 bits of global history and 14 bits of local history, that is, 55 weights [11]. We have analyzed different history setups exposed in [12] and, in general, 40–14 proved to be the most efficient for the evaluated table sizes. However, this data does not take into account the extra latency needed for the computation of the perceptron output. Con-

sequently, we add an extra cycle to the perceptron table latency shown in figure 4 to model this computation process. This is an optimistic assumption according to [12].

3.4. Branch Predictors Setup

We have evaluated the four simulated fetch engines varying the size of the branch predictor from small and fast tables to big and slow tables. Taking into account realistic table access latencies, the best performance is achieved using the larger three cycle latency tables [25]. Although bigger predictors are slightly more accurate, their increased access delay harms processor performance. On the other hand, predictors with a lower latency are too small and provide a poor performance. Therefore, we have chosen to simulate all branch predictors using the bigger tables that can be accessed in three cycles.

Table 2 shows the configuration of the simulated predictors, as well as the single-cycle predictors used by the overriding mechanism. We have explored a wide range of history lengths, as well as DOLC configurations for the trace and stream predictors, and selected the best one found for each setup. Table 2 also shows the approximated hardware budget for each predictor. Since we simulate the larger three cycle latency tables[1], the total hardware budget devoted to each predictor is different. The stream fetch engine requires less hardware resources because it uses a single prediction mechanism, while the other evaluated fetch architectures use two separate structures.

2bcgskew predictor (approx. 89KB)
four 64K entry tables 16 bit history (bimodal 0 bits) 4096 entry, 4-way associative, interleaved BTB
64 entry single-cycle gshare (6-bit history) 32 entry direct-mapped single-cycle BTB
perceptron predictor (approx. 46KB)
256 perceptrons 40 bit global and 4096x14 bit local history 4096 entry, 4-way associative, FTB
64 entry single-cycle gshare (6-bit history) 32 entry direct-mapped single-cycle BTB
next stream predictor (approx. 33KB)
1024 entry, 4-way associative, first level 4096 entry, 4-way associative, second level DOLC 16-2-4-10
32 entry direct-mapped single-cycle stream predictor DOLC 0-0-0-5
next trace predictor (approx. 74KB)
2048 entry, 4-way associative, first level 4096 entry, 4-way associative, second level DOLC 10-4-7-9 4096 entry, 4-way associative, interleaved BTB
32 entry direct-mapped single-cycle trace predictor DOLC 0-0-0-5 perfect BTB override

Table 2. Branch predictors setup.

4. The Length of Basic Prediction Units

Long basic prediction units help to hide the branch predictor access latency. The longer a prediction is, the more cycles the execution engine will be busy without requiring a new prediction. Figure 5 shows the average length of the basic prediction units for the four evaluated fetch architectures and for both the baseline and optimized code layouts.

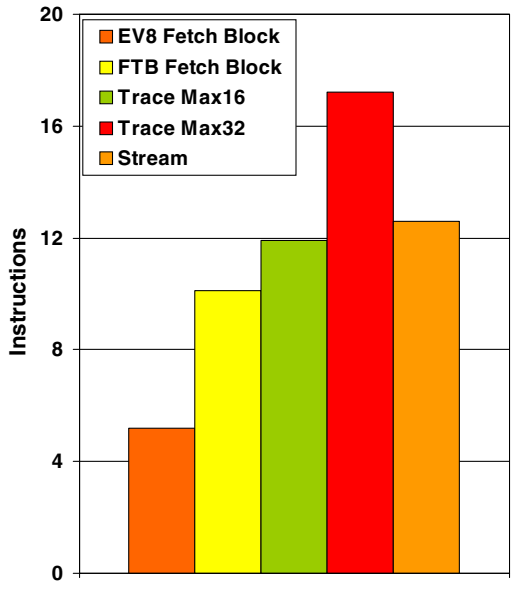
EV8 fetch blocks are the shorter prediction unit. They have the lower probability of hiding the branch predictor access by executing instructions from previous predictions. Therefore, our EV8-like fetch model requires an overriding mechanism to provide a high performance. FTB fetch blocks are longer, so they are less sensible to the branch predictor access latency. However, instruction streams are longer than both EV8 and FTB fetch blocks, so it can be expected that the stream predictor without overriding provides a high performance, even outperforming the FTB and EV8-like fetch architectures using overriding.

1 The first level of the trace and stream predictors is actually smaller than the second one because larger first level tables do not provide a significant improvement in prediction accuracy.

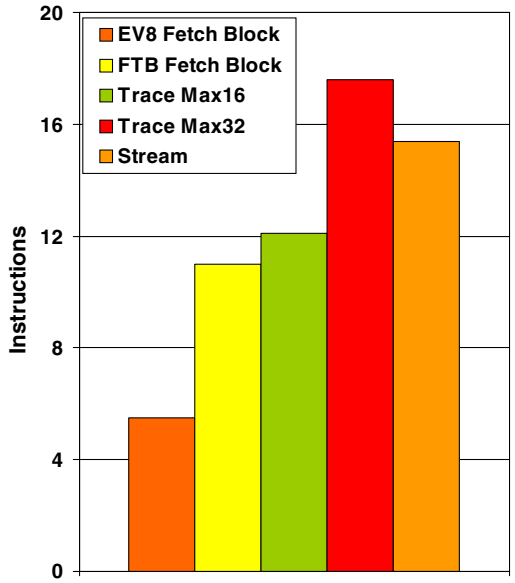
Figure 5 also shows that code layout optimizations have a beneficial effect on the length of instruction streams. Code layout optimizations try to map together those basic blocks which are frequently executed as a sequence. Therefore, most dynamic conditional branches in optimized codes are not taken, enlarging instruction streams. On average, instruction streams are 22% longer when using optimized codes, increasing the stream predictor ability of tolerating the access latency.

Instruction traces are also long enough to hide the predictor access latency. However, since traces are stored in a special purpose cache, their size is physically limited. Figure 5 shows data for both 16-instruction and 32-instruction maximum trace size. Increasing the maximum trace size involves an increase in the average size (although a trace is also limited by other factors, like indirect branches), but it will also reduce the total number of traces that can be stored in the trace cache, increasing its miss rate.

Although the 32-instruction setup has longer traces, it is able to store half the number of traces than the 16-instruction setup. We have found that the 32-instruction setup provides a better performance in the 8-wide proces-



(a) baseline code



(b) optimized code

Figure 5. Average basic prediction unit length in the four evaluated fetch engines.

sor. However, the 16-instruction setup provides a better performance in the 4-wide processor because a lower number of instructions is required to hide the predictor delay, being more beneficial to store a larger number of traces in the trace cache. Nevertheless, streams are long enough to provide a performance similar to a trace cache at a lower complexity [19], specially when using optimized codes.

5. Performance Evaluation

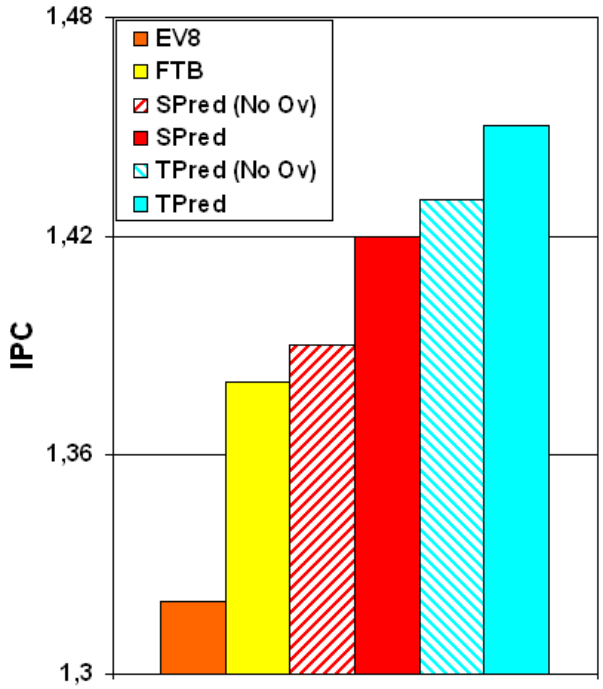
Figure 6 shows the average processor performance achieved by the four evaluated fetch architectures using the 4-wide setup. Data is shown for both the baseline and the optimized code layout. Besides the performance of the four fetch engines using overriding, the performance achieved by the stream fetch engine and the trace cache fetch architecture not using overriding is also shown (labeled *No Ov*).

When all evaluated models use overriding, that is, under equal conditions, the stream fetch engine and the trace cache fetch architecture outperform the EV8-like and FTB models. Moreover, the stream fetch engine and the trace cache fetch architecture do not need the complex overriding mechanism to outperform the EV8-like and FTB fetch architectures.

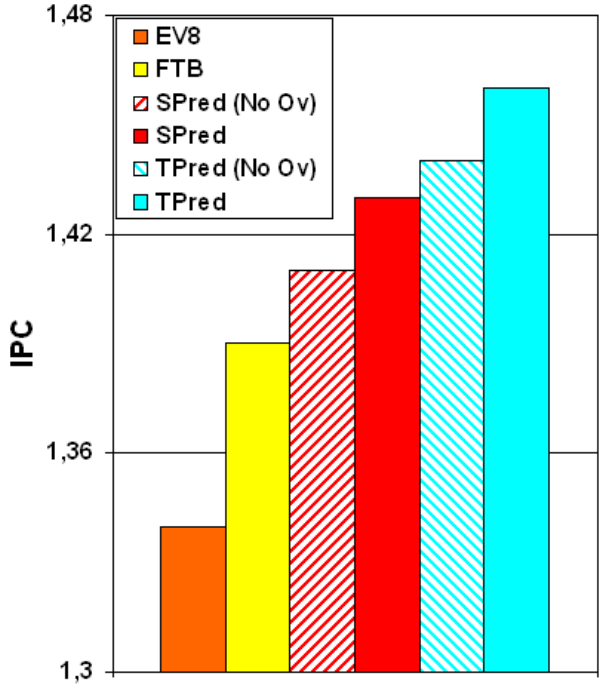
Using the baseline code layout, the stream fetch engine without overriding achieves an average IPC speedup of 5% over the EV8-like fetch engine using overriding. Code layout optimizations enlarge the average length of fetch blocks. Since EV8 fetch blocks are shorter than instruction streams, code optimizations are more beneficial for the EV8-like fetch architecture performance than for the stream fetch engine. However, using optimized codes, the stream fetch engine not using overriding still achieves an average 4% IPC speedup over the EV8 fetch architecture with overriding. Even more, if the EV8-like fetch architecture does not use overriding, the stream fetch engine would achieve an speedup over 40% (not shown in the figure). The trace cache fetch architecture not using overriding achieves speedups even higher than the stream fetch engine.

Since FTB fetch blocks are longer than EV8 ones, the FTB fetch architecture is less sensible to prediction delay. However, the stream fetch engine without overriding achieves a performance similar to an FTB fetch architecture using overriding. The stream predictor not using overriding achieves an average 0.5% IPC speedup against the FTB with overriding, which is raised to 1.5% when using optimized codes. If the FTB fetch architecture does not use overriding, the stream fetch engine would achieve an speedup higher than 5%. Once again, the trace cache fetch architecture not using overriding provides larger IPC speedups.

Figure 7 shows the average processor performance achieved by the four evaluated fetch architectures using the 8-wide setup. In this setup, a larger number of instructions is needed each cycle to keep the execution engine busy. This fact reduces the ability of the trace and stream predictors for tolerating the access latency. Consequently, the stream fetch engine without overriding is not able to outperform the FTB fetch architecture using overriding. It achieves an average 3.5% IPC slowdown. In spite of this, the stream fetch engine not using over-

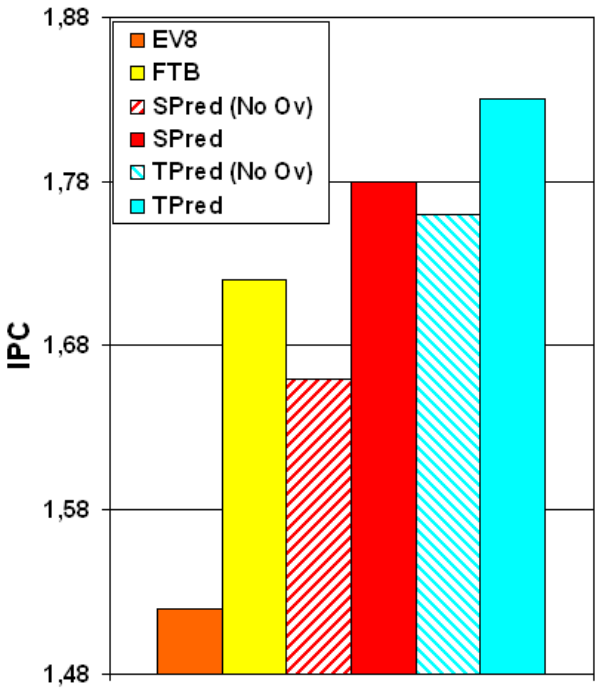


(a) baseline code

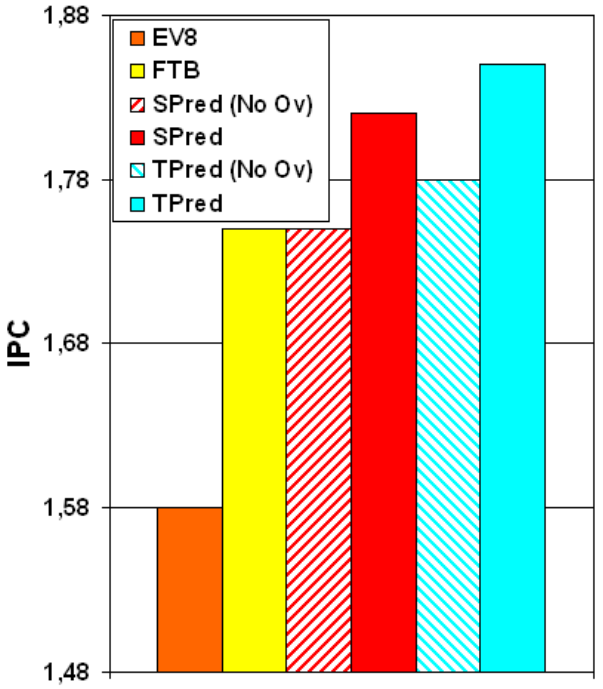


(b) optimized code

Figure 6. Average processor performance achieved by the 4-wide evaluated setups.



(a) baseline code



(b) optimized code

Figure 7. Average processor performance achieved by the 8-wide evaluated setups.

riding outperforms the EV8-like fetch architecture using overriding, achieving an average 9% IPC speedup. In addition, the trace cache fetch architecture not using overriding outperforms both the EV8-like and the FTB fetch architectures using overriding.

Using optimized codes, the longer size of streams compensates the need for a larger number of instructions in the 8-wide setup. The stream fetch engine not using overriding performs almost equally well than the FTB fetch architecture using overriding. Since the performance of both fetch architectures is so close, the stream predictor without overriding is a good choice in order to reduce the fetch engine complexity. In addition, the stream predictor without overriding still outperforms the EV8-like fetch engine using overriding, achieving an average 11% IPC speedup.

On average, the trace cache is the most efficient evaluated fetch mechanism, thanks to its ability of fetching instructions beyond taken branches in a single cycle. In addition, the trace predictor without overriding outperforms both the FTB and EV8-like fetch architectures using overriding in all the evaluated setups. Thus, a trace predictor not using overriding is also an interesting alternative to remove the overriding mechanism, still achieving a high performance. The main disadvantage of the trace predictor regarding the stream predictor is its higher complexity. It requires an special-purpose storage and a dynamic building mechanism, while instruction streams are sequentially stored in the instruction cache. Moreover, the stream fetch engine is only slightly slower than the trace cache. Using optimized codes, the IPC slowdown is even reduced under 2%. Therefore, the stream predictor represents the best tradeoff between processor performance and fetch engine complexity.

Nevertheless, both the stream and trace predictors are able to achieve a higher performance when using overriding. An overriding mechanism is specially complex for a trace predictor, since the secondary predictor used to build traces should be taken into account. Not only the instructions based on wrong predictions should be squashed, but also the portions of traces built based on wrong predictions should be undone. Since the stream predictor using overriding provides a performance similar to a trace predictor using overriding, it still represents the best tradeoff between processor performance and fetch engine complexity.

6. Conclusions

In modern superscalar processors, higher clock frequencies and larger wire delays cause branch prediction tables to require multiple cycles to be accessed. This is an important limiting factor for fetch engine performance, since branch predictions should be completed in a single cycle to allow fetching instructions in the following cycle. Prediction overriding is an efficient technique proposed to overcome this problem. However, it needs a complex recovery mechanism to discard the wrong speculative work based on overridden predictions.

In this paper, we have shown that the stream and trace predictors are able to tolerate their access latency without using such an overriding mechanism, reducing fetch engine complexity. Instruction streams and traces are long enough to feed the execution engine during multiple cycles, hiding the access delay of the following prediction. We have shown that both the stream and trace predictors without overriding are able to outperform other efficient fetch mechanisms, such as an EV8-like fetch architecture and the FTB fetch engine, even when they do use overriding.

It is true that, using prediction overriding, the stream fetch engine and the trace cache fetch architecture provide a better performance than not using it. This fact leads to an important decision in the design of the processor front-end: reducing the fetch engine complexity or increasing its performance using overriding. Nevertheless, it is still possible to enlarge streams and traces. If we obtain longer prediction units, the branch predictor access latency can be completely hidden, removing all the complexity of the overriding mechanism without loosing performance.

As we have shown, code layout optimizations enlarge instruction streams, increasing the ability of the stream predictor to tolerate the access latency. Therefore, more research effort should be devoted to new optimization techniques in order to achieve streams long enough to completely hide the access latency. In the same way, techniques to enlarge instruction traces would also be useful to hide the access latency of the trace predictor. In [22] some dynamic trace selection techniques are proposed in order to enlarge instruction traces. Another approach is the rePLay microarchitecture [15]. It uses a front-end derived from the trace cache, making extensive use of the branch promotion technique [14] to build very long instruction traces, called frames, and then dynamically optimize them.

In general, long instruction traces will avoid the need of an overriding mechanism in a trace cache fetch architecture. However, it does not remove the additional complexity of such architecture, which is its main disadvantage against the stream fetch engine. If performance is the only relevant factor, then the trace cache is probably the best option, since it provides other advantages like storing decoded instructions [7] or enabling dynamic optimizations [5, 15]. However, if complexity is also taken into account, the stream fetch architecture provides a worthwhile alternative.

Acknowledgements

This research has been supported by CICYT grant TIC-2001-0995-C02-01, an Intel scholarship grant, and CEPBA. Oliverio. J. Santana is also supported by Generalitat de Catalunya grant 2001FI-00724-APTIND.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

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

[3] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: an optimizer for Alpha/NT executables. *Proceedings of the USENIX Windows NT Workshop*, 1997.

[4] A. Falcon, O. J. Santana, A. Ramirez, and M. Valero. Tolerating Branch Predictor Latency on SMT. *Proceedings of the 5th International Symposium on High Performance Computing*, 2003.

[5] D. H. Friendly, S. J. Patel, and Y. N. Patt. Alternative fetch and issue techniques from the trace cache mechanism. *Proceedings of the 30th International Symposium on Microarchitecture*, 1997.

[6] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, vol. 10, no. 14, 1996.

[7] 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.

[8] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal useful logic depth per pipeline stage is 6-8 fo4. *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

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

[10] D. A. Jimenez, S. W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

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

[12] D. A. Jimenez. Delay-sensitive branch predictors for future technologies. *Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin*, 2002.

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

[14] S. J. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[15] S. J. Patel, T. Tung, S. Bose, and M. M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. *Proceedings of the 33rd International Symposium on Microarchitecture*, 2000.

[16] 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.

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

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

[19] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero. Fetching instruction streams. *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.

[20] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[21] R. Rosner, A. Mendelson, and R. Ronen. Filtering techniques to improve trace cache efficiency. *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[22] R. Rosner, M. Moffie, Y. Sazeides, and R. Ronen. Selecting long atomic traces for high coverage. *Proceedings of the 17th International Conference on Supercomputing*, 2003.

[23] E. Rotenberg, S. Benett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[24] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, vol 48, no. 2, 1999.

[25] O. J. Santana, A. Ramirez, J. L. Larriba-Pey, and M. Valero. Accurate Latency-Tolerant Branch Prediction. *Technical Report UPC-DAC-2003-09*, 2003.

[26] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the Alpha EV8 conditional branch predictor. *Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

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

[28] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[29] 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.

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

1527-1366/03 $ 17.00 © 2003 IEEE