

An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors

Pierre Michaud, André Seznec, Stéphan Jourdan

Abstract

The effective performance of wide-issue superscalar processors depends on many parameters, such as branch prediction accuracy, available instruction-level parallelism, and instruction-fetch bandwidth. This paper explores the relations between some of these parameters, and more particularly, the requirement in instruction-fetch bandwidth.

We introduce new enhancements to boost effectively the instruction-fetch bandwidth of conventional fetch engines. However, experiments strongly show that the performance does not increase as fast as the instruction fetch bandwidth. At the level of bandwidth exhibited by the proposed schemes, the performance improvement is small : once the measured IPC is half the instruction fetch bandwidth, increasing the fetch bandwidth brings very little improvement. This clearly brings to light potential relations between the fetch bandwidth and the other parameters.

In the second part of the paper, we develop a model from the empirical observation that the available instruction parallelism grows as the square root of the instruction window size. We show that this model is coherent with the main experimental observations. From the model, we derive that the fetch bandwidth requirement grows as the square root of the distance between mispredicted branches. We also verify experimentally that to double the IPC one should both double the fetch bandwidth and decrease the number of mispredicted branches fourfold.

1 Introduction

Out-of-order execution is widely used in superscalar processors. As tens of millions of transistors are now available, future processor generations are likely to feature large on-chip resource (e.g. caches, functional units).

To achieve high performance in a wide-issue superscalar processor, the fetch engine should supply instructions at a sufficient rate. Conventional fetch engines supplying a single sequential basic block are not capable of feeding such an aggressive microprocessor. Recently, new concepts have been proposed to implement high-bandwidth instruction fetch engines. Such concepts include the trace cache approach [14, 4] and the multiple-block fetching approach [19, 3, 16].

This paper proposes to characterize the extent to which increasing the instruction fetch bandwidth increases the performance. Our experimentations show that, assuming a fixed execution core, increasing the instruction fetch brings diminishing returns. If the available instruction parallelism in the application, assuming unlimited bandwidth, is X , then in practice a fetch bandwidth of $2X$ instructions per cycle is sufficient.

To better understand this result, we develop a model by focusing on three of the performance bottlenecks : the instruction-fetch rate, the branch prediction accuracy, and data dependencies. This model is based on the empirical observation that, for instruction windows less than 1k instructions, the instruction parallelism grows as the square root of the instruction window size. Our model shows that for doubling the performance, one should both double the fetch bandwidth and decrease the mispredict rate fourfold.

This paper is divided in two main parts. In Section 2, we propose a simple solution to boost the effective fetch rate of conventional instruction fetch engines. Our best engine supplies around 13 instructions per cycle. However this very same engine does not provide a significant performance improvement, even using an execution core with no resource limitations. We show experimentally that this is due to the limitations from both the available parallelism and the branch predictor performance. Section 3 introduces an analytical model to quantify such interactions. Section 4 provides some concluding remarks.

2 Performance Impact of Instruction Fetch Mechanisms in Out-of-Order Superscalar Processors

Two solutions have been proposed to implement high-bandwidth instruction-fetch engines. The conventional approach relies on regular instruction caches and branch predictors [19, 3, 16]. The other option is based on a *trace cache* [14].

In this section, we focus on high-bandwidth conventional engines. We describe and then compare four different fetch engines of increasing hardware complexity and bandwidth. The four designs with the corresponding bandwidths are:

- the usual *one block-ahead* (OBA) scheme fetching one basic block,
- the E-OBA scheme fetching up to two basic blocks when the first block ends in a not-taken branch,
- the *two block-ahead* (TBA) scheme [16] fetching any two basic blocks,
- the E-TBA scheme, which fetches up to four basic blocks when the first and third branches are both not taken.

The purpose of this section is two-fold. We introduce two new instruction fetch engines, namely E-OBA and E-TBA, and emphasize that E-TBA delivers a high instruction bandwidth. On the other hand, we show that processor performance does not scale with such high bandwidth.

2.1 High-Bandwidth Instruction Fetching

Boosting bandwidth in conventional designs requires work mainly in branch prediction. Instruction caches are less a problem since techniques like banking or phase pipelining, where structures are pipelined over half cycles like in the data cache of the Alpha 21264, provide an efficient way to support such high bandwidths. Branch prediction is basically a sequential process. Using the conventional OBA scheme, a straightforward way to predict two blocks per cycle is to make prediction tables small enough so that two successive accesses can be performed in a single cycle. However, this solution impairs prediction accuracy.

The TBA scheme is an alternative. It overcomes the sequential aspect by performing the prediction not with the block containing the predicted branch, but with the block immediately before. By doing so, consecutive predictions are overlapped. The TBA scheme, through dual porting, banking, or phase pipelining, predicts up to two basic blocks in a single cycle while keeping prediction tables large.

We propose to further increase the instruction bandwidth delivered by the TBA scheme by allowing basic blocks to extend through one not-taken branch, introducing the extended TBA (E-TBA) scheme. This technique can be applied also to a conventional OBA scheme, defining the extended OBA (E-OBA) scheme. All these schemes are described in the remainder of this section.

2.2 The E-OBA Scheme: Bypassing One Not-taken Branch

The conventional OBA scheme wastes instruction bandwidth since, on a predicted not-taken branch, the instructions following the branch in the fetch window are discarded and fetched again in the next cycle. Ideally, one would want to bypass all not-taken branches, as was proposed in [3]. Instead of fetching basic blocks, we would fetch sequential traces, a sequential trace ending on the first taken branch. Though this approach may require widening the fetch window, the instruction cache remains single-ported.

Bypassing all not-taken branches requires interleaving the branch prediction tables (e.g. the BTB and the two-bit counter table) on as many banks as the number of instructions in the fetch window (the fetch window can be a cache line, two adjacent cache lines, two adjacent half-lines ...). This may not be cost-effective compared to a conventional OBA mechanism.

The solution we propose is based on the observation that bypassing only a single not-taken branch brings most of the benefits of fetching full sequential traces. On average in the IBS traces, 30 % of the dynamic branches are not-taken conditional branches (75 % of the branches are conditional with a 40 %

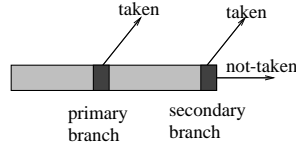


Figure 1: An extended block: when the primary branch is not-taken, the block is extended up to the secondary branch

not-taken ratio). When bypassing all not-taken branches, the average fetch rate is 1.4 ($= 1/0.7$) basic blocks per cycle. When bypassing only one not-taken branch, the average fetch rate is 1.3 basic block per cycle. As a result, bypassing one not-taken branch is almost equivalent to fetching complete sequential traces.

We define an *extended block* as a basic block extended past a single not-taken branch as shown in Figure 1. Whenever the first branch encountered (the *primary branch*) is not-taken, the block is extended up to the next branch (the *secondary branch*).

The implementation of the E-OBA scheme is depicted in Figure 2. It works exactly like a regular OBA scheme, but with extended blocks instead of basic blocks. We assume *fetch address based indexing* [20]: the address of block N is used to predict primary branch N and secondary branch N+1 when branch N is not-taken. The prediction tables are split into two arrays, in order to deliver a primary and a secondary prediction simultaneously. E-OBA does not feature specific *primary* and *secondary* arrays: this would not distribute branches evenly on the arrays since, in our benchmarks, 77 % ($= 1/1.3$) of branches are primary branches. Instead, we decide which array is the primary array on the basis of the block address (for example, in Figure 2, we use bit a_5 of the fetch address because line boundaries may bias lower-order address bits).

The E-OBA scheme delivers on average 1.3 basic blocks per cycle, not taking into account the fetch window limitations. Two basic blocks are fetched simultaneously when the first basic block ends on a not-taken branch. Further increase of the fetch bandwidth requires fetching two basic blocks in any case. This is the purpose of the TBA scheme.

2.3 The Basic TBA Scheme [16]

The basics of the TBA scheme are depicted in Figure 3. In summary, a branch is identified with the address of the previous block and the direction of the previous branch. For instance in Figure 3, block N address is used to predict branch N+1 and generate the address of block N+2. Similarly, block N+1 is used to generate the address of block N+3. Both address generations are performed in parallel.

The TBA scheme is completely symmetric, meaning that predictions do not rely on a particular pairing of blocks. This symmetry allows to use many existing branch prediction schemes, e.g. bimodal, global, hybrid, or de-aliased. Furthermore, prediction bandwidth can be obtained either by banking or by pipelining [16].

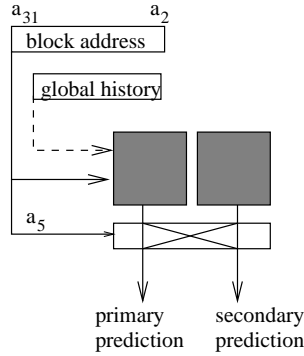


Figure 2: E-OBA scheme: the secondary prediction is used if the primary branch is not-taken

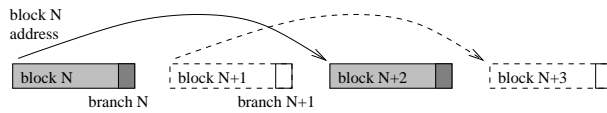


Figure 3: Principle of TBA prediction

As described in [16], branch predictor structures consist in a *branch target buffer* to identify branches, a 2-bit counter table to predict conditional branch outcomes, a tagged *target cache* [2] to predict indirect branches, a *return address stack* to predict returns, and a *prediction stack* to identify and predict the first branch following a return.

All the tables are organized as depicted in Figure 4. Tables are split into two arrays, to allow reading predictions without knowing the direction of the previous branch initially. The direction, once known, is used to select the output. Since taken branches outnumber not-taken branches, we XOR the direction with an address bit to distribute entries evenly over the two arrays.

Further details are provided in [16], especially the use of the prediction stack (the branch after a return is predicted as if the original *call* were not-taken), the mechanism to restart the fetch process after a misprediction, and how to take into account blocks larger than the fetch window.

2.4 The E-TBA Scheme

In order to further increase the bandwidth provided by the TBA scheme, we can modify it to work on extended blocks in place of basic blocks, defining an E-TBA scheme. This means predicting the address of the next extended block using the previous extended block. In E-TBA, there are three ways to exit an extended block:

- T_1 : taken primary branch,

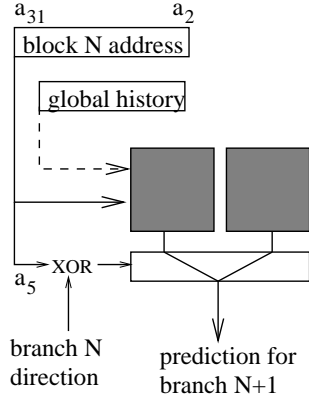


Figure 4: TBA indexing on a 2-bit counter table, a BTB, or a target cache

- T_2 : not-taken secondary branch,
- T_3 : taken secondary branch.

Extended-block transitions can be coded using 2 bits: the first bit specifies whether we exit the extended block on a primary or a secondary branch, and the second bit specifies the direction of the branch (one single bit records the transition in the regular TBA scheme). A special exit condition occurs when the extended block exceeds the fetch window. In this particular case, we exit on a line boundary and the transition can be coded as T_2 without ambiguity.

Figure 5 provides a **logical** view of the E-TBA prediction tables (e.g. BTB, target cache, 2-bit counters). The tables are split into three array pairs. These six arrays are accessed simultaneously with the address of extended-block N. Each pair delivers a primary and secondary prediction to generate the address of extended-block N+2. The transition T_N at the end of extended-block N is then used to select one of three predictions.

Practically, in this implementation, the table space would not be used evenly since primary branches outnumber secondary branches, and transition T_1 occurs more often than T_2 and T_3 . It is better to split the table into four array pairs. The two bits of transition T_N are XORed with two address bits to select one of four array pairs. Another address bit indicates which array in the selected pair delivers the primary and secondary prediction respectively.

As for the TBA scheme, these structures must be multi-ported or pipelined to provide two predictions in a single cycle.

2.5 Experimental Evaluation of OBA, E-OBA, TBA and E-TBA

All experiments in this section are conducted with a trace-driven simulator using the IBS traces [18]. The eight traces reflect the execution of sequential

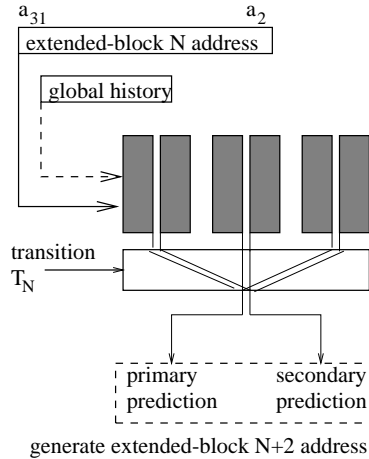


Figure 5: E-TBA scheme (logical)

applications on a MIPS-based workstation, including system activity.

Processor model. We modeled an aggressive superscalar out-of-order processor with some optimistic assumptions to emphasize the impact of instruction fetching. The processor issues up to 16 instructions from a window of 256 instructions. It features 16 uniform pipelined units and a 4-port data cache. Data cache misses are not simulated. Instruction latencies are those of the COMPAQ/DEC Alpha 21264. Loads are issued as soon as their memory address is computed, even when previous store addresses are not known yet. When a store matches an already issued load, the load and the instructions dependent on it are selectively re-issued.

The instruction pipeline is 8-stage long: *fetch*, *align*, *decode*, *rename*, *dispatch*, *read operands*, *execute*, and *retire*. Instructions are retired from the instruction window in the program order, and instruction fetching on the correct path resumes as soon as a mispredicted branch is executed. All stages but the fetch stage are 16-instruction wide.

Fetch mechanisms. The instruction cache is 256 Kbytes. Lines are 16-instruction wide and the miss latency is 8 cycles. For the OBA and E-OBA schemes, the fetch window is made of two adjacent lines. For the TBA and E-TBA schemes, the fetch window is made of two adjacent half-lines. In any case, a maximum of 32 instructions can be fetched in a cycle. We use a 64-instruction *fetch buffer* between the align and decode stages, where instructions wait for decode when more than 16 instructions are fetched in the same cycle (otherwise, the fetch buffer is bypassed).

The branch predictor consists in a 16k-entry BTB, a 3x16k-entry *e-gskew*

[11] to predict conditional branches, a 4k-entry tagged *target cache*, and a *return address stack*. The BTB and the target cache are cascaded [5] to predict indirects branches. The global history length is 12 bits.

The TBA and E-TBA schemes also feature a *prediction stack*. We slightly modified the scheme described in [16] to benefit from the target cache. The target cache can help the BTB to identify a branch following an indirect branch. The target cache and BTB entries have the same format, and both prediction tables are used in a cascaded manner. Simulations showed that the use of a target cache decreases significantly the number of branches misidentified after an indirect branch.

Simulation results. Table 1 shows the IPC values (instructions retired per cycle) and fetch rates (valid instructions fetched per fetch cycle on the right path) measured on the simulated processor. First we observe that the techniques used to increase the fetch bandwidth are very effective. Fetch rates achieved by E-TBA are between 11 and 16 instructions per cycle, with an average close to 13 : it is roughly twice the fetch rate offered by the OBA scheme. In fact, we fetch less than 2.6 basic blocks per cycle because of the fetch window limitations and mispredicted branches (we do not count instructions fetched after a mispredicted branch)

The measured misprediction rate is nearly the same for the four schemes. Table 2 shows the average number of basic blocks executed per mispredicted branch (conditional or indirect). The rightmost column gives the ratio between E-TBA and OBA. The most important drop in prediction accuracy is 10 % on *verilog*.

As expected, higher fetch rates improve performance. However, the performance improvement is not proportional to the increase in fetch bandwidth. While the extra 25 % bandwidth of E-OBA over OBA brings a speedup of 1.13, the extra 40 % bandwidth of TBA over E-OBA returns only a speedup of 1.12, and the extra 20 % bandwidth of E-TBA over TBA only returns a speedup of 1.02. From OBA to E-TBA, the fetch rate doubles but the gain in IPC is only 25-30 %.

The diminishing performance return would be even more dramatic had we simulated a realistic memory hierarchy. To analyze the processor behavior, we broke simulation cycles into:

- **fetch cycles:** valid instructions are fetched
- **decode cycles:** a misidentified or mispredicted branch has been fetched and is waiting to be decoded
- **misprediction cycles:** a mispredicted branch has been decoded and is waiting to be executed
- **full-buffer cycles:** the fetch buffer is full of *valid* instructions

Results are reported in Figure 6, measured in cycles per instruction (CPI). With the E-TBA scheme, the gain on fetch cycles is compensated by more

| | | OBA | E-OBA | TBA | E-TBA | E-TBA / OBA |
|------------|-----------|------|-------|-------|-------|-------------|
| groff | IPC | 4.00 | 4.60 | 5.06 | 5.18 | 1.29 |
| | fetch | 5.50 | 7.05 | 9.98 | 11.96 | 2.17 |
| | IPC/fetch | 0.73 | 0.65 | 0.51 | 0.43 | |
| gs | IPC | 4.14 | 4.76 | 5.32 | 5.42 | 1.31 |
| | fetch | 5.62 | 7.20 | 10.26 | 12.23 | 2.18 |
| | IPC/fetch | 0.74 | 0.66 | 0.52 | 0.44 | |
| mpeg_play | IPC | 5.05 | 5.54 | 6.03 | 6.14 | 1.22 |
| | fetch | 7.90 | 9.91 | 13.46 | 15.89 | 2.01 |
| | IPC/fetch | 0.64 | 0.56 | 0.45 | 0.39 | |
| nroff | IPC | 3.93 | 4.89 | 6.34 | 6.80 | 1.73 |
| | fetch | 4.52 | 5.90 | 8.74 | 10.94 | 2.42 |
| | IPC/fetch | 0.87 | 0.83 | 0.73 | 0.62 | |
| real_gcc | IPC | 3.53 | 3.92 | 4.28 | 4.35 | 1.23 |
| | fetch | 5.69 | 7.43 | 10.13 | 12.32 | 2.17 |
| | IPC/fetch | 0.62 | 0.53 | 0.42 | 0.35 | |
| sdet | IPC | 4.39 | 4.69 | 5.19 | 5.24 | 1.19 |
| | fetch | 6.49 | 7.67 | 12.08 | 13.88 | 2.14 |
| | IPC/fetch | 0.68 | 0.61 | 0.43 | 0.38 | |
| verilog | IPC | 4.11 | 4.59 | 4.87 | 4.94 | 1.20 |
| | fetch | 6.00 | 7.50 | 10.05 | 11.61 | 1.94 |
| | IPC/fetch | 0.68 | 0.61 | 0.48 | 0.43 | |
| video_play | IPC | 5.10 | 5.75 | 6.51 | 6.65 | 1.30 |
| | fetch | 6.69 | 8.32 | 12.02 | 14.17 | 2.12 |
| | IPC/fetch | 0.76 | 0.69 | 0.54 | 0.47 | |

Table 1: IPC values and average fetch rates on the simulated processor

| | OBA | E-OBA | TBA | E-TBA | E-TBA / OBA |
|------------|-------|-------|-------|-------|-------------|
| groff | 51.39 | 50.92 | 48.74 | 48.33 | 0.94 |
| gs | 42.89 | 41.24 | 40.48 | 39.90 | 0.93 |
| mpeg_play | 30.08 | 30.00 | 29.62 | 29.67 | 0.99 |
| nroff | 59.07 | 58.92 | 58.17 | 58.44 | 0.99 |
| real_gcc | 19.91 | 18.91 | 18.72 | 18.40 | 0.92 |
| sdet | 42.85 | 42.48 | 42.21 | 42.94 | 1.00 |
| verilog | 43.86 | 43.04 | 40.06 | 39.52 | 0.90 |
| video_play | 55.63 | 54.12 | 51.91 | 51.85 | 0.93 |

Table 2: Average number of basic blocks executed per mispredicted branch on the simulated processor

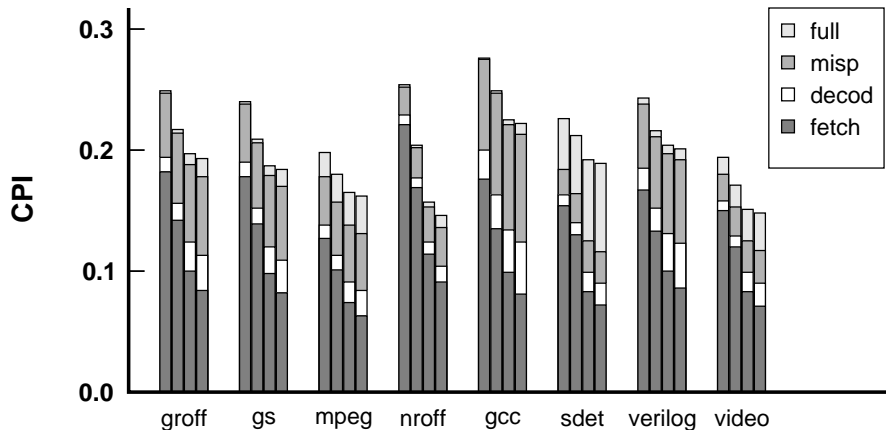


Figure 6: CPI breakdown (bars from left to right represent OBA, E-OBA, TBA and E-TBA respectively)

misprediction cycles and decode cycles. The number of misprediction cycles increases since instructions enter the instruction window sooner, so they wait longer for data dependencies. The number of decode cycles also increases because the decode and rename width (16 instructions) is too small compared to the fetch width. Though a moderate-size fetch buffer is sufficient to regulate the flow of instructions (as the average fetch rate keeps below 16 instructions per cycle), it introduces several *virtual* pipeline stages where instructions wait for idle decode slots. This problem, which is specific to E-TBA, could be alleviated by enlarging the decode and rename widths.

2.6 Summary

Our experiments have shown that bypassing one not-taken branch is an efficient technique to increase the fetch bandwidth. The E-TBA scheme delivers on average 13 instructions per cycle. Note that this bandwidth gain is proportional to the ratio between taken and not-taken branches. The extended-block schemes become even more effective when taking into consideration compiler techniques to increase the proportion of not-taken branches, as in [1].

We have also shown that, even with a very aggressive model (for today's standards), the performance return of increasing the instruction fetch bandwidth becomes lower: while E-TBA delivers 20 % more bandwidth over TBA, results show a 1.02 performance speedup only.

The next section is an attempt to explain why increasing the fetch bandwidth is not always beneficial to performance.

3 A model of the interactions between instruction fetching, branch prediction, and ILP

The goal of this section is to provide a better understanding of the performance of a superscalar processor in relation with the instruction fetch rate and branch prediction accuracy. We first observe that, empirically, the instruction-level parallelism (ILP) grows as the square root of the instruction window size. We build a simple analytical model of the execution based on this assertion. We show that this model is able to reproduce the main conclusions of the previous section, highlighting the relations between instruction fetching, branch prediction accuracy, and ILP.

3.1 Methodology and processor model

In this section, we focus on three of the performance bottlenecks in the superscalar paradigm : data dependencies, the distance between control flow breaks, and the instruction fetch rate (we use the term *control flow breaks* instead of *branch mispredictions* because it is more general).

All experiments in this section are conducted with a very simple simulator that models only the studied parameters. The processor modeled is an idealized out-of-order superscalar processor with no resource constraints, no front-end pipeline stages, and a perfect bypass network between functional units : instructions fetched are ready for execution as soon as their operands are available. Basically, the simulator processes one instruction at a time, computing the execution cycle as

$$exec_cycle = \max(fetch_cycle, \{availability_cycle\}_{operands}) + exec_latency$$

All instructions but loads and stores have an execution latency of one cycle. Loads and stores use one cycle for the memory address computation and one or several cycles for accessing the cache. We assume all accesses hit in the cache. Otherwise stated, the cache access latency is one cycle. We assume a perfect memory disambiguator so that loads do not have to wait longer than necessary.

Throughout this section, the term **slice** refers to a sequence of dynamic valid instructions between two consecutive control flow breaks. All the slices pasted together form the whole instruction trace. On a control flow break, we assume the processor waits for all the previously fetched instructions to be executed before resuming the fetch of valid instructions. So a new slice cannot be fetched until all the instructions of the previous slice have been executed.

3.2 The square root law

We define a perfect fetch engine as an engine able to fetch one slice in a cycle, since it allows to minimize the execution time of each slice and therefore of the whole application. With a perfect fetch engine, and assuming no resource

constraints, the execution of a slice can be represented as a **data-flow graph** : each instruction in the slice is a node of the graph, and an instruction B is connected to a previous instruction A if B uses the result of A. Each node is labeled with the execution latency of the corresponding instruction (we could also label edges with bypass latencies, but as we assume a perfect bypass network, edge latencies are null). The length of a chain in the data-flow graph is computed as the sum of the latencies of all the instructions on the chain, and the execution time of the slice is equal to the length of the longest chain in the graph.

To evaluate the average length of the longest chain in a slice, we partition the instruction trace into slices of constant size S . We define L_S as the length of the longest chain in a slice averaged on all the slices in the benchmark trace. This provides statistical information on the structure of the data-flow graph in programs. Figure 7 shows the value of L_S measured on the IBS and SPECint95 benchmarks when S varies from 8 to 4k instructions. The cache latency is one cycle on the left graphs and 4 cycles on the right graphs. We use a logarithmic scale on both axis of the graphs.

First, it can be observed that the majority of our benchmarks have very similar behaviors. With a cache latency of 1 cycle,

$$L_S = \sqrt{2S}$$

is a good approximation of the average length of the longest data-flow chain in slices less than $S=1k$ instructions. For larger slices, the experimental curves diverge from the model. Among our benchmarks, only IBS *sdet* does not follow the square root law : this benchmark spends a lot of time in a very tight loop, so, unsurprisingly, it does not follow the average behavior. We also observe that when we increase the cache latency, the experimental curves keep following a square root law. In the remaining, we will assume a generalized square root law

$$L_S = \frac{1}{\alpha} \sqrt{S}$$

Parameter α quantifies the efficiency of the execution core and depends on many factors : operators latencies, bypass latencies, number of data cache misses, cache miss latency, use of data value prediction, ... The higher α is, the faster the processor drains the instruction window.

Another way to observe the square root law is to simulate a processor having a reorder buffer with a perfect fetch engine keeping it always full, as we did in [12] : the IPC varies according to the square root of the reorder buffer size W . We show, in the appendix, the following inequality :

$$\alpha\sqrt{W} \leq IPC \leq 2\alpha\sqrt{W}$$

It should be noted that Riseman and Foster observed a similar square root law almost 3 decades ago [13]. The square root law seems to accurately model the data flow of a large set of applications, and especially applications having a complex control flow.

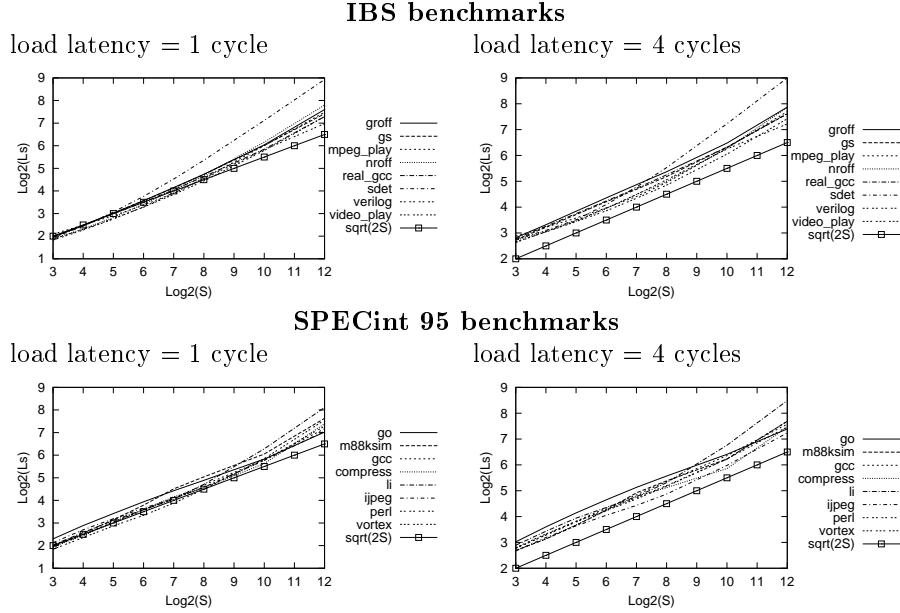


Figure 7: Value of L_S on the IBS and SPECint95 benchmarks when S varies from 8 to 4096

3.3 Impact of a limited fetch width : defining milestone instructions

Now let us focus on a single slice of size S , and let us call L_S the length of the longest chain in the data-flow of this slice. When the whole slice is brought in the instruction window instantaneously, the time to execute the slice is equal to L_S cycles. On the other hand, if it takes several cycles to bring the whole slice in the window, this may delay the execution of some instructions and increase the overall execution time. To take this into account, we modify the data-flow graph by introducing new edges corresponding to fetch cycle constraints.

Figure 8 shows an example of the data-flow of a slice of 13 instructions. Instructions are numbered sequentially from 1 to $S = 13$. To take into account fetch cycle constraints, we add a node 0 in the graph (node 0 can be viewed as the previous slice) and we connect each instruction i in the slice to node 0, labeling each new edge with the fetch cycle of the instruction, $f(i)$. We call this new graph \mathcal{G}_f (see Figure 8).

The execution time of the slice can be computed as the length of the longest path in graph \mathcal{G}_f . As node 0 is connected to all the other nodes, node 0 is indeed on the longest path, so we only need to consider the paths starting from node 0. The maximum path length can be computed as

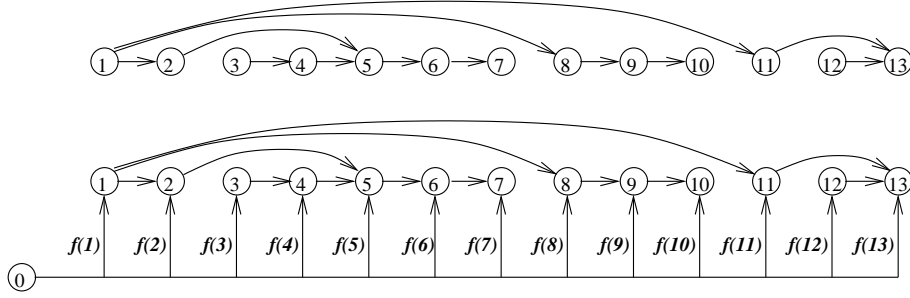


Figure 8: Upper graph : pure data-flow. Lower graph : graph \mathcal{G}_f with new edges to take into account fetch cycle constraints

$$T = \max_{i=1 \dots S} (f(i) + l(i))$$

with $l(i)$ being the length of the longest chain originating from instruction i toward the end of the slice. Figure 9 gives the $l(i)$ values on the example data-flow.

In principle, we must know $l(i)$ for each instruction i in the slice to model the fetch. However, it is possible to simplify the problem by considering that instructions are fetched in the program order. In other words, the fetch function f is an increasing function. If we consider the set of instructions in the slice that have the same value $l(i) = n$, the maximum of $f(i) + l(i)$ on these instructions is determined by the instruction fetched the latest. We call this instruction the **milestone** instruction of rank n :

$$\forall n \in [1 \dots L_S], \quad m_n = \max\{i \in [1 \dots S] / l(i) = n\}$$

The milestone instructions on the example of Figure 9 correspond to the shaded $l(i)$ values. The expression for the execution time becomes

$$T = \max_{n=1 \dots L_S} (f(m_n) + n)$$

To compute T , we only need to know the number of milestones in the slice and the time at which each milestone is fetched.

More fetch bandwidth is needed at the beginning of the slice : It is possible to minimize T by fetching one milestone per cycle

$$\forall n \in [1 \dots L_S], \quad f_{opt}(m_n) = L_S - n$$

This is an “optimal” fetch, as we try in each cycle to fetch the minimum number of instructions without delaying the execution time. Actually, the distribution of milestones in the data flow of programs is not uniform. Milestones are distributed sparsely in the beginning of the slice, and they get closer to each other

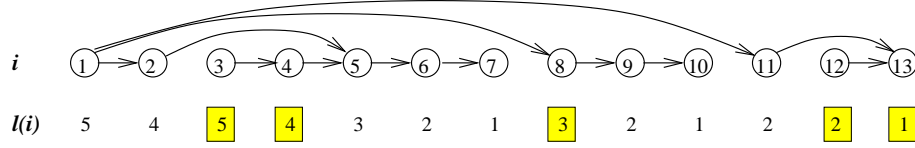


Figure 9: Example of a slice data-flow with 5 milestones

near the end of the slice. Consequently, the optimal fetch is not uniform : a large fetch bandwidth is needed at the beginning of the slice, and the fetch bandwidth requirement decreases gradually as the fetch proceeds.

However, existing fetch mechanisms can be better modeled with a uniform fetch, so we assume a uniform fetch in the remaining.

3.4 The milestone model

For modeling the impact of a limited fetch, we must have a model for $f(m_n)$. To keep the model simple, we assume a uniform fetch width F , with a fetch function $f(i) = i/F$, and we model $f(m_n)$ as

$$f(m_n) = \frac{S - \alpha^2 n^2}{F}$$

The rationale behind this function is that the number of milestones within a given distance from the end of the slice, on average, follows the square root law. Now the execution time can be expressed as

$$T = \max_{1 \dots L_S} \left(\frac{S - \alpha^2 n^2}{F} + n \right)$$

The shape of $h(n) = \frac{S - \alpha^2 n^2}{F} + n$ is represented in Figure 10. We can approximate T by computing the maximum of the parabola $h(x)$ between 1 and L_S . The critical milestone is the milestone m_n for which $h(n)$ is maximum, and its rank is $F/(2\alpha^2)$:

$$T \approx \frac{S}{F} + \frac{F}{4\alpha^2}$$

This is true as long as $F \leq 2\alpha^2 L_S$. The critical milestone rank increases with F . When $F > 2\alpha^2 L_S$, the critical milestone rank is maximum and equal to L_S . Considering $L_S = \frac{1}{\alpha} \sqrt{S}$, when $F > 2\alpha \sqrt{S}$, T is minimum and equal to $T_{min} = \frac{1}{\alpha} \sqrt{S}$

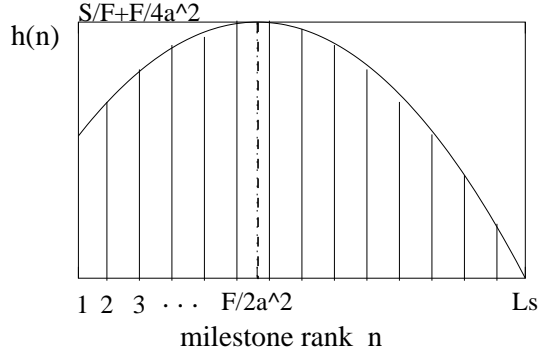


Figure 10: $h(n) = \frac{S - a^2 n^2}{F} + n$

Existence of a threshold fetch The execution time is minimum when the fetch is greater than

$$F_t = 2\alpha\sqrt{S}$$

We call this value the **threshold fetch**, as there is no benefit to increase the fetch over this value. When $F \leq F_t$, the execution time is the sum of the fetching time S/F , which is the time to fetch all the instructions up to the next control flow break, plus the draining time $F/(4\alpha^2)$, which is the time for the instruction provoking the control flow break to retire from the processor.

When F is small, the fetching time dominates the execution time : the performance is limited by the fetch. When F is large, the draining time increases, the gain on the fetching time is compensated partly by the draining time. Beyond the threshold fetch, the performance is limited by the execution core. The performance in number of valid instructions executed per cycle may be computed as S/T :

$$F \leq F_t, \quad IPC \approx \frac{F}{1 + (F/F_t)^2}$$

$$F > F_t, \quad IPC = \frac{F_t}{2}$$

It can be noticed that the threshold fetch for a slice is reached when the fetch width is the double of the highest possible IPC in this slice. Figure 11 shows the curves for the IPC predicted by the milestone model for slice sizes $S = 50, 100, 200$.

Consequence on the fetch engine tuning Considering branch mispredictions as the main source of control flow breaks, there is a direct relation between

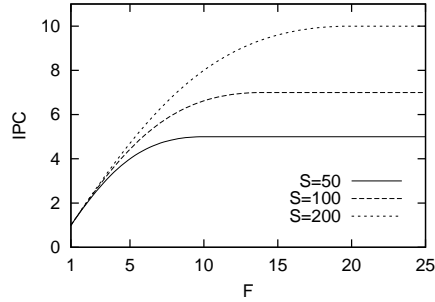


Figure 11: IPC given by the milestones model assuming $\alpha = 1/\sqrt{2}$

the average size of a slice and the branch misprediction ratio : if we divide the misprediction ratio by a factor k , we multiply the average slice size by the same factor k . We observe that the curves on Figure 11 are homothetic : considering a fixed α , to double the IPC, we must at the same time double the fetch width and decrease the mispredict ratio fourfold.

On the other hand, our model shows that trading branch prediction accuracy to get more instruction bandwidth is only worthwhile when the fetch bandwidth is under-dimensioned, but would be counterproductive whenever the fetch bandwidth is already close to the threshold. When the threshold fetch is reached, we should rather try to improve the branch prediction accuracy or increase α .

3.5 A variant : the leaky-bucket model

The model presented in [12] is a more intuitive model based on an analogy with a leaky bucket : the velocity of the water flowing out of the bucket is $\sqrt{2gh}$, h being the height of water in the bucket. We can model the instruction window of the processor as a bucket into which the fetch engine pours instructions and which is drained simultaneously by the execution core. We call C_t the number of instructions remaining to be fetched up to the end of the slice. We call N_t the number of instructions waiting for execution in the instruction window in cycle t , and we assume that the number of instructions that can execute in this cycle is $\sqrt{2gN_t}$, g being a “gravity” constant representing the efficiency of the execution core. We get the following algorithm :

$$\begin{aligned}
 N_0 &= 0 \\
 C_0 &= S \\
 N_{t+1} &= N_t - \sqrt{2gN_t} + \min(C_t, F) \\
 C_{t+1} &= C_t - \min(C_t, F)
 \end{aligned}$$

The algorithm ends when $N_t + C_t \leq 0$. Figure 12 plots $\sqrt{2gN_t}$ in each cycle, for a slice size of 200 instructions and for different fetch widths. The plot exhibits clearly three ILP phases : in the first cycles, the ILP rises (the window is filling),

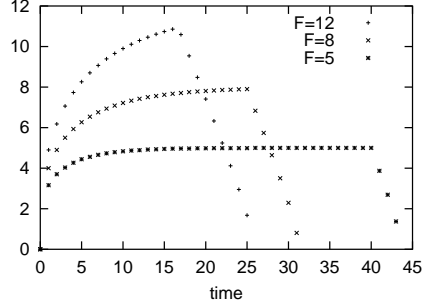


Figure 12: Bucket model : plot of $\sqrt{2gN_t}$ in each cycle, for a slice $S=200$ and for different fetch widths

then it stabilizes at a constant ILP equal to the fetch width, and when the whole slice has been fetched, the ILP falls (the window is emptying). For small fetch widths, the constant ILP phase dominates the execution time : a gain on the fetch is a gain on the execution time. As the fetch width increases, the constant ILP phase gets shorter : the gain on the fetch is compensated partly by a longer falling ILP phase. The threshold fetch is reached when the constant ILP phase vanishes : the execution time is then limited by the execution core.

Assuming the constant ILP phase is non null, we can calculate the execution time by noting that the slope of the falling ILP phase is nearly constant :

$$\begin{aligned}
 \sqrt{2gN_{t+1}} - \sqrt{2gN_t} &= \sqrt{2gN_t} \left(\sqrt{1 - \frac{\sqrt{2g}}{N_t}} - 1 \right) \\
 &\approx \sqrt{2gN_t} \left[\left(1 - \frac{1}{2} \sqrt{\frac{2g}{N_t}} \right) - 1 \right] \\
 &\approx -g
 \end{aligned}$$

The execution time can be expressed as the time to fetch all the instructions in the slice plus the falling ILP time

$$T \approx \frac{S}{F} + \frac{F}{g}$$

We find an expression similar to the one given by the milestones model, assuming $g = 4\alpha^2$.

3.6 Model limitations and experimental verification

This modeling effort was primarily aimed at explaining the experimental results of section 2. We modeled only on a few (but important) processor parameters.

In particular, we did not take into account resource constraints and the pipeline depth, though it could be possible to extend the leaky-bucket model. We also focused on a conventional superscalar paradigm, so the model may not be adapted to other paradigms, like those exploiting control independence.

Nevertheless, the model allows to understand how instruction fetching and branch prediction interact. It is able to explain the empirical observation that, once the fetch rate is the double of the measured IPC, increasing the fetch brings no significant performance improvement. The model also predicts that, to double the IPC, we should both double the fetch width and decrease the mispredict rate fourfold.

To verify this, we returned to the experimental set-up of section 2. To emulate a better branch predictor than in Section 2.5, we removed some branch mispredictions in a random fashion. The processor configuration is almost the same as in Section 2.5, except that, to scale with a higher branch prediction accuracy, we increase the processor instruction window size from 256 to 1024, and the pipeline width from 16 to 32.

Figure 13 shows the IPC on the new processor configuration with the base branch predictor (upper graph) and when 75 % of mispredictions are removed (lower graph). The performance differences between the four fetch engines increase when we decrease the number of mispredictions. The OBA scheme exhibits a 1.2 performance speedup when 75% of mispredictions are removed, whereas E-TBA speedup tops at 1.35. We verify that the new IPC of E-TBA is approximately twice the IPC of OBA with 4 times more mispredicted branches.

4 Summary and Concluding Remarks

In this paper, we emphasized the relations between the instruction-fetch bandwidth, the available ILP, and the branch prediction accuracy.

First, we introduced a simple solution to boost the instruction-fetch bandwidth of a conventional instruction cache. By allowing a block to continue beyond one not-taken branch, the E-TBA scheme gets 20 % extra instruction-fetch bandwidth over a TBA scheme, at a modest hardware cost. However, this extra bandwidth was not found to be really useful on a superscalar processor executing a single instruction flow and constrained by true data dependencies.

For a better understanding of the experimental results, we focused on data dependencies, branch mispredictions and the instruction fetch bandwidth. We developed a model based on the empirical observation that the available instruction parallelism grows as the square root of the instruction window size. This model corroborates the main experimental observations. It highlights the existence of a threshold instruction-fetch rate beyond which little performance gain is to be expected. According to the model, the threshold fetch grows as the square root of the distance (in instructions) between two consecutive mispredicted branches. The threshold fetch rate is approximately twice the maximum IPC that can be obtained with an unlimited fetch. We also verified that to double the performance, we should both double the fetch rate and decrease four-fold

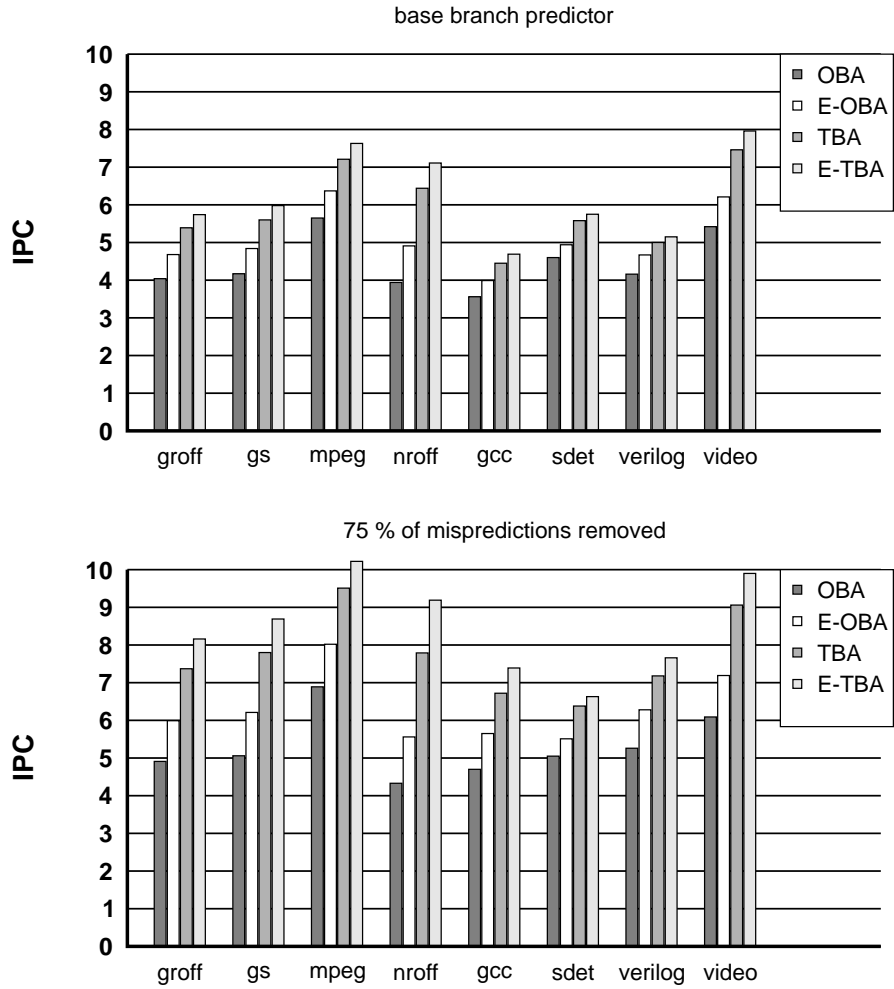


Figure 13: Impact on the IPC of removing 75 % of mispredictions

the number of mispredicted branches.

The conclusion on the tuning of the instruction-fetch engine is that when the threshold fetch is reached, we should rather try to decrease the number of control flow breaks [10, 7, 8] or improve the execution core [9, 15, 17, 6].

Finally, we would like to conclude on the methodology in computer architecture research. Most microarchitectures studies use simulators modeling a complete processor as accurately as possible, that is, modeling all mechanisms present in a real processor or already proposed elsewhere. However, in most studies focusing on a particular point of the processor design, e.g. instruction fetch, all other parameters cannot be completely explored. Simulating “true” mechanisms, but not varying their parameters, leads to biased simulation results. From our experiment in Section 2.5, one may disregard the impact of E-TBA based on the poor performance improvement reported. Had we modeled techniques to boost the amount of ILP or to increase the distance between mispredictions, the conclusion would be different.

This leads us to argue for deemphasizing complete processor simulation, and to focus on more significant metrics than just the overall performance of a simulated processor. For instance, the average fetch rate and the branch misprediction rate characterize an instruction fetch engine better than the IPC value obtained on a fixed execution core.

Appendix

We consider a processor with a reorder buffer of size W and a perfect fetch engine keeping it always full. We start from the basic data-flow graph \mathcal{G} of the whole dynamic instruction trace.

We consider a graph \mathcal{G}' derived from \mathcal{G} by adding *window dependence* edges between every pair of instructions separated by at least W instructions in the dynamic instruction trace (an instruction can enter the reorder buffer only after the instruction which is W -instruction ahead has been retired from the reorder buffer). The total execution time is given by the length $L_{\mathcal{G}'}$ of the longest chain in \mathcal{G}' . Let us consider the chain in \mathcal{G}' formed by partitioning the whole trace into N slices of W instructions, then concatenating the longest local data-flow chains in one every two slices (that is, slices 1,3,5,7,...). Asymptotically, the length of this chain is $\frac{N}{2}L_W$, so

$$L_{\mathcal{G}'} \geq \frac{N}{2}L_W$$

As $IPC = \frac{NW}{L_{\mathcal{G}'}}$, we have

$$IPC \leq \frac{2W}{L_W}$$

Now let us consider the graph \mathcal{G}'' derived from \mathcal{G} by adding edges between every pair of instructions belonging to different slices. The longest chain in \mathcal{G}'' is the chain formed by concatenating the longest local data-flow chains in each slice,

so $L_{G''} = NL_W$. Moreover, G' can be derived from G'' by removing some edges. As removing edges cannot increase the length of the longest chain,

$$L_{G'} \leq L_{G''}$$

consequently,

$$IPC \geq \frac{W}{L_W}$$

Eventually, considering $L_W = \frac{1}{\alpha}\sqrt{W}$, we have

$$\alpha\sqrt{W} \leq IPC \leq 2\alpha\sqrt{W}$$

References

- [1] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.
- [2] P.-Y. Chang, E. Hao, and Y.N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [3] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.
- [4] Keith Diefendorff. Hal makes Sparcs fly. *Microprocessor Report*, 13(15), November 1999.
- [5] Karel Driesen and Urs Holzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.
- [6] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.
- [7] A. Klauser, T. Austin, D. Grunwald, and B. Calder. Dynamic hammock predication for non-predicated instruction set architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1998.
- [8] Artur Klauser, Abhijit Paithankar, and Dirk Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

- [9] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit with value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.
- [10] Scott A. Mahlke, Richard E. Hank, Roger A. Bringmann, John C. Gyllenhaal, David M. Gallagher, and Wen mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.
- [11] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [12] Pierre Michaud, André Seznec, and Stéphan Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings the 1999 International Conference on Parallel Architectures and Compilation Techniques*, October 1999.
- [13] Edward Riseman and Caxton Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computer Architectures*, C-21(12):1405–1411, December 1972.
- [14] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.
- [15] Y. Sazeides, S. Vassiliadis, and J.E. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.
- [16] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [17] A. Sodani and G.S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [18] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.
- [19] T.-Y. Yeh, D.T. Marr, and Y.N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.
- [20] Tse-Yu Yeh and Yale Patt. Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors. In *Proceedings of the 26th International Symposium on Microarchitecture*, 1993.