Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors

Pierre Michaud, André Seznec, Stéphan Jourdan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

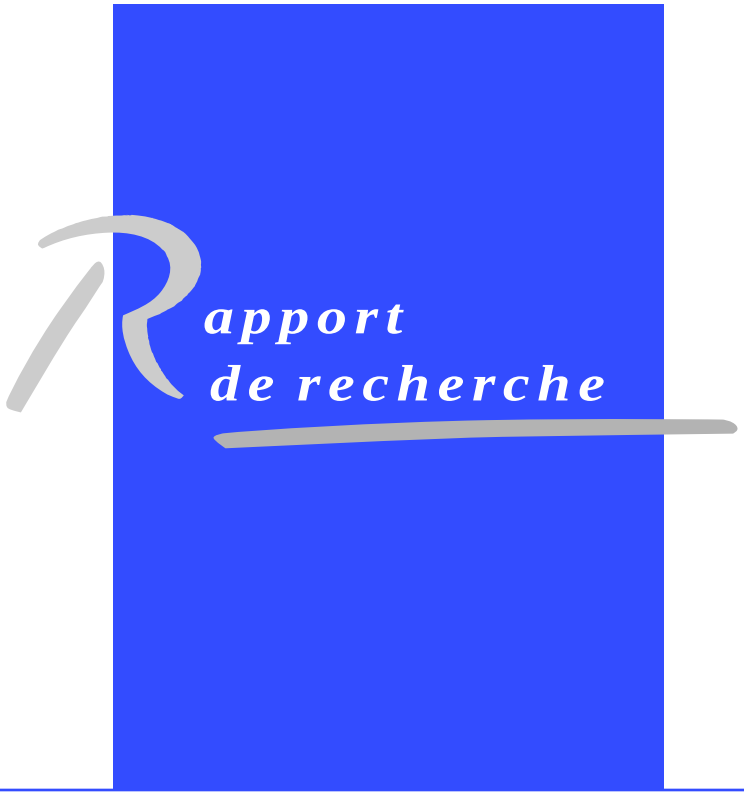
Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors

Pierre Michaud, André Seznec, Stéphan Jourdan

N ° 3604

Janvier 1999

THÈME 1



ISSN 0249-6399

Exploring Instruction-Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors

Pierre Michaud, André Seznec, Stéphan Jourdan *

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n3604 — Janvier 1999 — 23 pages

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 performance improves less for a given instruction-fetch bandwidth gain as the base bandwidth increases. At the level of bandwidth exhibited by the proposed schemes, the performance improvement is small. This clearly brings to light potential relations between the bandwidth and the other parameters. We provide a model to explain this behavior and quantify some relations. Based on the experimental observation that the available parallelism in an instruction window grows as the square root of the window size, we derive from the model that the instruction fetch bandwidth requirement increases as the square root of the distance between mispredicted branches. We also show that the instruction fetch bandwidth requirement increases linearly with the parallelism available in a fixed-size instruction window. Finally, we review some existing techniques to enhance performance and we describe their impact on the instruction-fetch requirement in the light of the above relations. These techniques include those increasing the amount of instruction-level parallelism (e.g. value-prediction) and those enlarging the effective instruction window (e.g. eager execution).

Key-words: superscalar processors, instruction fetch, branch prediction, instruction-level parallelism, square root model

(Résumé : tsvp)

* Intel Corporation

Unit´e de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
T´el´ephone : 02 99 84 71 00 - International : +33 2 99 84 71 00
T´el´ecopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Exploration du besoin en chargement d'instructions sur les processeurs superscalaires

Résumé :

La performance effective des processeurs superscalaires dépend de nombreux paramètres, comme le taux de branchements mal prédits, le parallélisme d'instructions disponible, et le débit de chargement d'instructions. Ce papier explore la relation entre certains de ces paramètres, et plus particulièrement les besoins en débit de chargement d'instructions. Nous introduisons des améliorations pour augmenter le débit des mécanismes classiques de chargement d'instructions. Cependant, nos expériences montrent que le gain de performance apporté par les mécanismes introduits est marginal sur les processeurs superscalaires classiques. Cela suggère une relation entre le débit de chargement et d'autres paramètres. Nous introduisons un modèle pour expliquer ce comportement et quantifier les relations existantes. En observant expérimentalement que le parallélisme d'instructions augmente comme la racine carrée de la taille de la fenêtre d'instructions, nous déduisons que le débit de chargement requis augmente comme la racine carrée de l'inverse du taux de branchements mal prédits. Nous montrons également que le débit de chargement requis est proportionnel au parallélisme d'instructions disponible dans une fenêtre d'instructions de taille fixe. Enfin, nous passons en revue des techniques récentes pour améliorer les performances des processeurs et montrons, à la lumière des relations précédentes, l'impact de ces techniques sur les besoins en débit de chargement. Parmi ces techniques, nous distinguons celles augmentant le parallélisme d'instructions (par exemple, la prédiction de valeur) et celles augmentant la fenêtre d'instructions effective (par exemple, l'exécution dédoublée).

Mots-clé : processeurs superscalaires, chargement d'instructions, prédiction de branchement, parallélisme d'instructions, modèle en racine carrée

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). Because of this increased complexity, the performance depends on many factors such as the available instruction-level parallelism, the branch prediction accuracy, and the effective instruction-fetch rate.

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. Recent research studies have proposed new concepts to implement high-bandwidth instruction fetch engines. Such concepts include the trace cache approach [26, 23, 8] and the more conventional multiple block fetching approach [33, 6, 28, 32].

Relations between specific parameters are seldom analyzed in academic studies. For instance, this is the case between the effective fetch rate, the branch prediction accuracy, and the available parallelism. Computer architects usually rely on some rules of thumb such as "the higher the fetch bandwidth, the higher the performance" or "the available instruction-level parallelism increases with the branch prediction accuracy". This paper proposes to study the interactions between the three parameters listed above.

To achieve this, we first enhance, in Section 2, conventional instruction-fetch engines to boost the effective fetch rate. Our best engine is capable of supplying around 13 instructions in a single cycle. We then show that this very same engine does not provide a significant performance improvement. 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. Relations are explicitly stated. Finally, we review in Section 4 some existing techniques to enhance performance and we describe their impact on the instruction-fetch requirement in the light of the above relations. These techniques include those increasing the amount of instruction-level parallelism (e.g. value-prediction), and those enlarging the effective instruction window (e.g. predication). Section 5 provides some concluding remarks.

Experimental Set-Up. All experiments in this study are conducted on the IBS traces [30]. The eight traces reflect the execution of sequential applications on a MIPS-based workstation, including system activity.

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 predictors [33, 6, 28, 32]. The other option is based on a *trace cache* [26, 23, 8].

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 fetching up to two basic blocks when the first block ends in a not-taken branch,
- the *two block-ahead* (TBA) scheme [28] 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 very high instruction bandwidth. On the other hand, we show that processor performance does not scale as well with such high bandwidth results.

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 DEC 21264 [11], 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 also be applied 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 [6]. Instead of fetching basic blocks, we would fetch sequential traces,

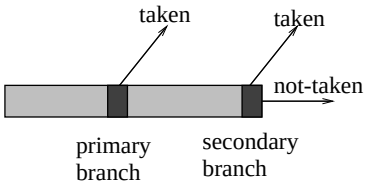


Figure 1: **An extended block: when the primary branch is not-taken, the block is extended up to the secondary branch**

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 % not-taken ratio). When bypassing all not-taken branches, the average fetch rate is 1.4 (= 1/0.7) basic blocks per cycle. By 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 on extended blocks instead of basic blocks. We assume *fetch address based indexing* [34]: 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 on average in the IBS benchmarks, 77% (= 1/1.3) of the branches are primary branches. Instead, we decide which array is the primary array on the basis of the block address.

The E-OBA scheme delivers on average 1.3 basic blocks in a single 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. Increasing the fetch bandwidth requires to fetch two basic blocks in any case. This is the purpose of the TBA scheme.

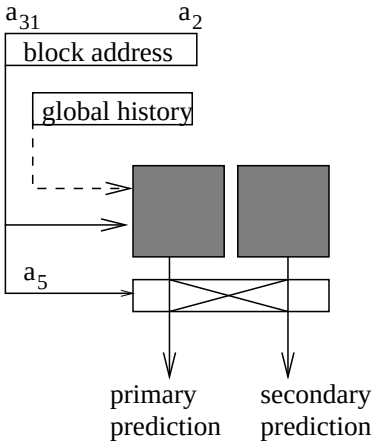


Figure 2: **E-OBA scheme: the secondary prediction is used if the primary branch is not-taken**

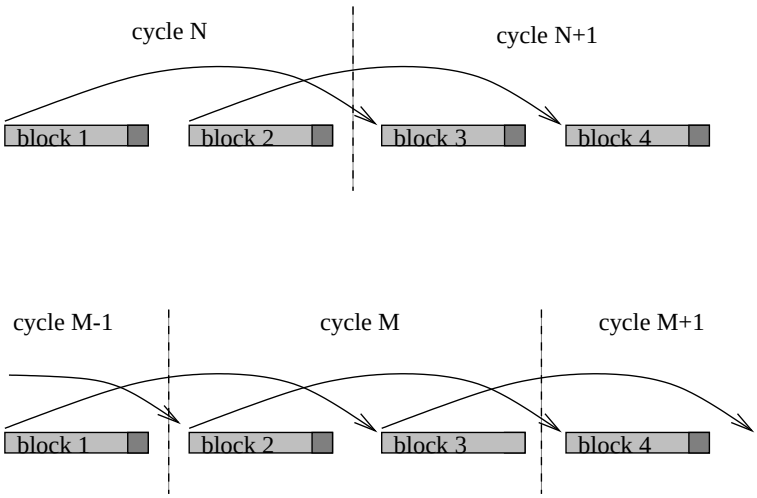


Figure 3: **Principle of TBA prediction**

2.3 The Basic TBA Scheme [28]

The basics of the TBA scheme are depicted on 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 1 is used to predict the branch associated to block 2 and generate the

address of block 3. Similarly, block 2 is used to generate the address of block 4. 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. Figure 3 gives two examples where blocks 3 and 4 are generated in the same cycle in one case, and in different cycles in the other case. In both examples, block 3 is generated in the exact same way. This symmetry enables to use any of the existing branch prediction schemes in TBA, e.g. bimodal, local, global, hybrid, or de-aliased. Furthermore, branch prediction can be done using dual-ported structures or can be pipelined [28]. This is of particular interest when using phase-pipelined instruction cache.

Brief description. As described in [28] , branch predictor structures consist in:

- a *branch target buffer* to identify branches,
- 2-bit counter tables to predict conditional branch outcomes,
- a tagged *target cache* [3] to predict indirect branches,
- a *return address stack* to predict returns,
- and a *prediction stack* to identify and predict the first branch following returns.

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 [28], especially the use of the prediction stack (the branch after a return is predicted as if the original *call* was 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,
- T_2 : not-taken secondary branch,
- T_3 : or taken secondary branch.

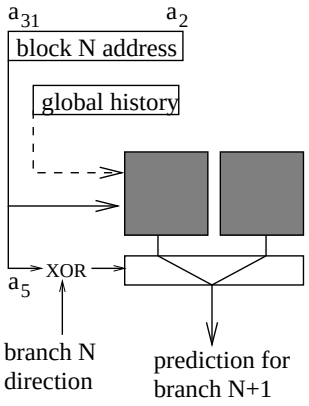


Figure 4: **TBA indexing on a 2-bit counter table, a BTB, or a target cache**

Extended block transitions can be coded using 2 bits: the first bit specifies if we exit the 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 is when the 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.

Indexing. 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 block N. Each pair delivers both a primary and a secondary prediction to generate the address of block N+2. The transition T_N at the end of block N is then used to select one of the three predictions.

Practically, this implementation of the tables is not efficient since primary branches outnumber secondary branches, and transition T_1 occurs more often than T_2 and T_3. Figure 6 describes a more cost-effective solution where the table is split into four array pairs. The two transition bits T_N are combined with two address bits to select one of the four pairs. Another address bit is then used to reorder primary and secondary predictions.

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

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-ported data-cache. Data cache misses are not simulated. Instruction latencies are those of the COMPAQ/DEC Alpha 21264. We assume a perfect memory

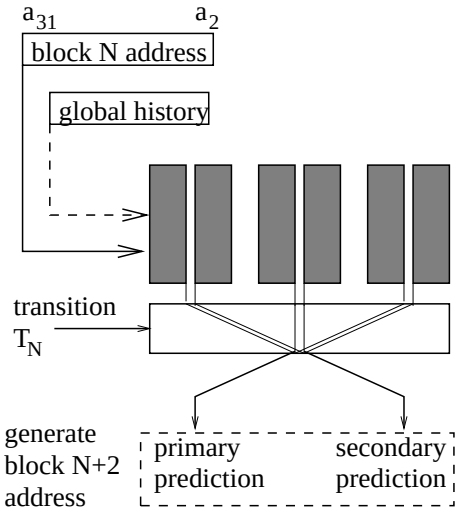


Figure 5: **E-TBA scheme (logical)**

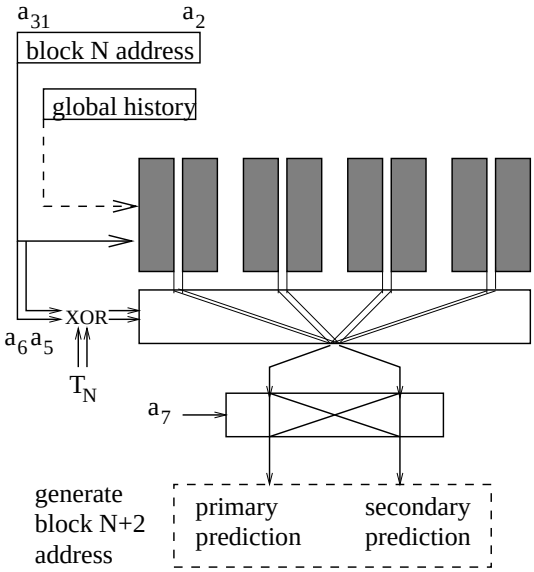


Figure 6: **E-TBA scheme (physical)**

disambiguator where non-colliding loads are issued as soon as their memory addresses are computed, even when previous store addresses are not known yet.

The instruction pipeline is 8 stages long: *fetch*, *align*, *decode*, *rename*, *dispatch*, *read operands*, *execute*, and *retire*. Instructions are retired from the window in-order, and execution from the correct path resumes as soon as mispredicted branches are resolved. The width of all the stages from *decode* to *execute* is 16 instructions.

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 both cases, up to 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* [20] to predict conditional branches, a 4k-entry tagged *target cache*, and a *return address stack*. The BTB and the target cache are cascaded [7] 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 [28] 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. Figures 7 and 8 detail IPC results and fetch rates (number of valid instructions fetched per fetch cycle) achieved by our model. The techniques used to increase the fetch bandwidth are very effective. Indeed, the TBA and E-TBA schemes have almost twice the fetch capacity of the OBA and E-OBA schemes respectively. There is only a small loss due to the limitations of the fetch window and mispredicted branches. Fetch rates achieved by E-TBA are between 11 and 16 instructions per cycle, with an average close to 13.

The measured misprediction rate (not plotted) is nearly the same for the four schemes: there is only a minor degradation in prediction accuracy going from OBA to E-TBA.

As can be seen in Figure 7, higher fetch rates always improves 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. Worst, the extra 20 % bandwidth of E-TBA over TBA only returns a speedup of 1.02. The diminishing performance return would be even more dramatic had we simulated a realistic memory hierarchy.

Why E-TBA performance is so disappointing? To better understand why the performance of E-TBA is so disappointing, 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 renamed, dispatched, and executed)
- full-buffer cycles (the fetch buffer is full).

With the E-TBA scheme, a large part of the gain on fetch cycles is lost in more 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, can be solved 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 scheme becomes even more effective when taking into consideration compiler techniques to increase the number of not-taken branches as in [2].

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 bandwidth is not always beneficial to performance. In Section 4, we give further explanation on how superscalar processors can take better advantage of the extra instruction fetch bandwidth delivered by TBA and E-TBA.

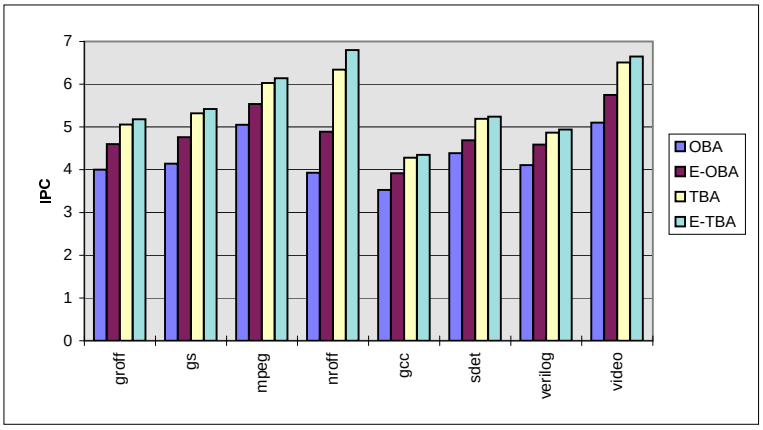


Figure 7: **Measured IPC**

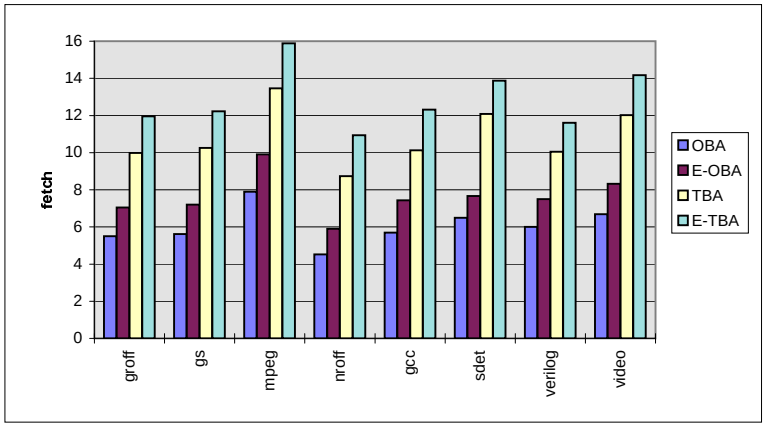


Figure 8: **Average fetch**

3 Relations between Instruction Fetching, Branch Prediction Accuracy, and Parallelism

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. In summary, we show that there is a threshold fetch rate which grows roughly as the square root of the distance between mispredicted branches and is proportional to the available instruction-level parallelism (ILP) in a fixed-size instruction window.

3.1 ILP as a Function of the Instruction Window Size

The impact of instruction fetching on performance depends on the way instructions are consumed. It is intuitive that when there is little ILP to extract, the processor does not need much instruction fetch bandwidth.

It is well known that ILP grows with the instruction window size [25, 31]. In order to quantify this growth, we have simulated a processor with no resource limitation but a fixed-size instruction window. We assume a perfect instruction fetch engine with perfect branch prediction, so that the instruction window remains always full. The execution of instructions is only constrained by true data dependencies.

Figure 9 illustrates the result of this experiment. The average ILP is given as a function of the instruction window size on a logarithmic scale. All IBS traces but *sdet* have an ILP that is approximately the square root $\sqrt{N}$ of the instruction window size N. This is an average law, and applications spending most of execution time in a small loop (as IBS *sdet* does) are less likely to fit the model.

It is interesting to note that Riseman and Foster found a similar $\sqrt{N}$ law [25].

3.2 Defining a $\sqrt{N}$ Model

In order to capture the basic relations between instruction fetch rate, misprediction rate, and the extractable ILP, we develop a model based on the $\sqrt{N}$ law and a very few additional assumptions. Although being too simple to capture real system behavior, such a model provides some insight by isolating the relevant factors.

In each cycle i, N_i is the number of valid instructions waiting for execution in the instruction window, and $\sqrt{N_i}$ instructions are executed in this cycle ($\sqrt{N_i}$ may not be an integer value, but this is just a model). The frequency of mispredicted branches is constant, and is defined by the number M of instructions between two mispredicted branches. F instructions are fetched in each cycle. We stop fetching upon a mispredicted branch. The fetch resumes when all instructions preceding the mispredicted branch have been executed. This defines a periodic behavior where the period is the constant time between mispredictions. In a period, cycle 0 is the cycle when fetching resumes on the correct path. Let C_i be the number of instructions remaining to be fetched up to the next mispredicted branch. The simulation algorithm is:

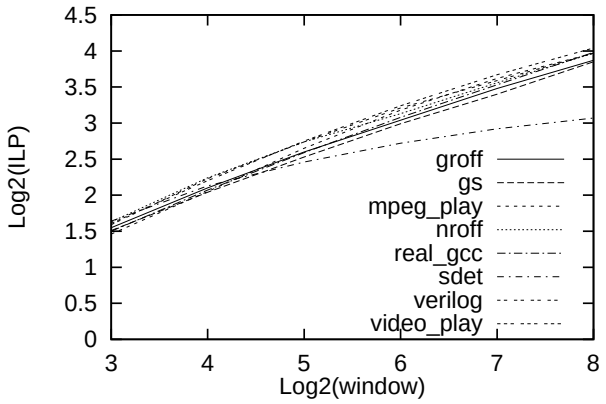


Figure 9: **Average ILP as a fonction of the instruction window size (logarithmic scale)**

$$N_0 = 0$$
$$C_0 = M$$
$$N_{i+1} = N_i - \sqrt{N_i} + min(C_i, F)$$
$$C_{i+1} = C_i - min(C_i, F)$$

The algorithm ends when $N_i + C_i \leq 0$. The average IPC, which measures performances, is the ratio between M and the number of simulation cycles.

3.3 Existence of a Threshold Fetch Rate

With this $\sqrt{N}$ model, we bring to light a threshold effect on the fetch rate: below the threshold, the performance increases quasi-linearly with the instruction fetch rate, whereas above the threshold, it grows much slower.

Figure 10 gives the average performance for M= 50, 100 and 200 instructions when the fetch rate varies between 1 and 16 instructions fetched per cycle. Increasing values of M correspond to increasing branch prediction accuracies. The diminishing performance return of increasing the fetch rate, observed in Section 2, appears clearly. For M=50 instructions, it is not worth fetching more than 4 instructions per cycle. For M=200, the threshold is around 8 instructions fetched per cycle. As foreseeable, large fetch bandwidths are more useful with accurate branch predictors.

Figure 11 shows the evolution of $ILP = \sqrt{N_i}$ cycle by cycle, assuming M=200 instructions, and for different fetch rates (the area below one curve is equal to M instructions).

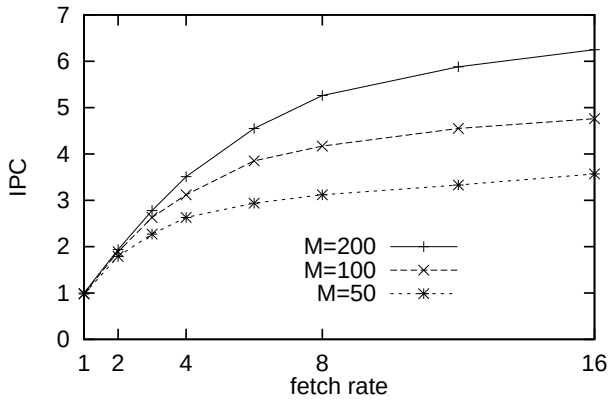


Figure 10: **IPC in function of the fetch rate for M=50,100 and 200 instructions on a $\sqrt{N}$ model**

The execution time can be divided into three phases: rising ILP, constant ILP, and falling ILP. The falling ILP phase starts when all M instructions have been fetched.

When the fetch rate is high (F=200), the falling ILP phase dominates the execution time: the execution rate is limited by the available ILP. When the fetch rate is low (F=4), the constant ILP phase dominates and the ILP value equals the fetch rate F: the execution rate is limited by the fetch rate. When the constant ILP phase dominates, a gain on the fetch rate is a gain on the execution time.

As the fetch rate increases, the constant ILP phase diminishes: the fetch rate for which the constant ILP phase vanishes is the threshold fetch rate (near 8 instructions fetched per cycle, on Figure 11). Beyond this threshold, increasing the fetch rate brings instructions sooner in the instruction window, but the shorter rising ILP phase is nearly compensated by a longer falling ILP phase.

3.4 The threshold fetch rate grows as the square root of the distance between mispredicted branches

As Figure 10 shows, the threshold fetch rate increases with M. We will not try to give precise values for the threshold fetch rate, as such values depend on many processor parameters. Our goal is only to quantify the growth of the threshold fetch rate with M.

We can evaluate the growth of the threshold fetch rate with M by noting (on Figure 11) that the slope of the falling ILP phase is nearly constant:

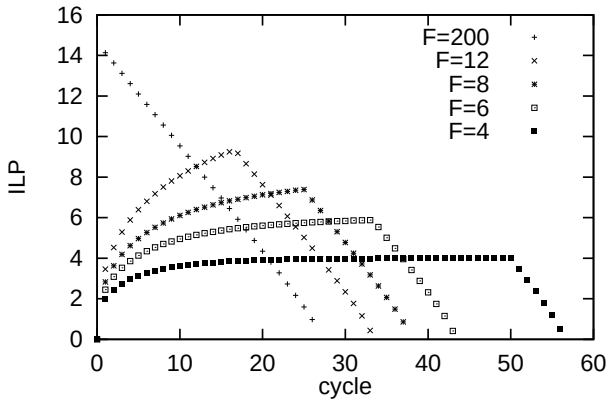


Figure 11: **Dynamic ILP** $= \sqrt{N_i}$ **for** M**=200**

$$\begin{aligned}\sqrt{N_{i+1}} - \sqrt{N_i} &= \sqrt{N_i}(\sqrt{\tfrac{N_i - \sqrt{N_i}}{N_i}} - 1) \\ &\simeq \sqrt{N_i}((1 - \tfrac{1}{2}\tfrac{1}{\sqrt{N_i}}) - 1) \\ &\simeq -\tfrac{1}{2}\end{aligned}$$

The time T_{min} to execute the M instructions when all M instructions are fetched in one cycle is

$$T_{min} \simeq 2\sqrt{M} \text{ cycles}$$

Then, the maximum average IPC is

$$IPC_{max} = \frac{M}{T_{min}} \simeq \frac{1}{2}\sqrt{M}$$

When the fetch rate is below the threshold, the average IPC can be obtained by noting that when the falling IPC phase starts (on cycle $\frac{M}{F}$), the previous phase was a constant $ILP = F$ phase. As a result, it takes $2F$ cycles to consume the F^2 instructions remaining in the window at that time:

$$IPC \simeq \frac{M}{\frac{M}{F} + 2F}$$

We assume that the threshold is reached when IPC is a fixed ratio of IPC_{max}:

$$\frac{M}{\frac{M}{F} + 2F} = K\frac{1}{2}\sqrt{M}$$

Solving this equation gives a threshold fetch rate proportional to $\sqrt{M}$. Curves on Figure 10 are homothetic. One conclusion of this is, to double the performance, we should increase the distance between mispredicted branches four-fold and double the instruction fetch rate.

3.5 The threshold fetch rate keeps proportional to ILP

Recent techniques, like data-value prediction, have shown that it is possible to collapse data dependencies. Let us take the assumption that we can extract more parallelism than in the $\sqrt{N}$ model. We assume we can issue $k\sqrt{N_i}$ instructions per cycle, with $k > 1$. The simulation algorithm becomes

$$N_{i+1} = N_i - k\sqrt{N_i} + min(C_i, F)$$

and we have

$$IPC_k(F, M) = k^2 IPC_1(\frac{F}{k^2}, \frac{M}{k^2})$$

The curves can be deduced from Figure 10 by multiplying IPC values, F values and M values by k^2. We showed that the threshold F for $IPC_1(F, M)$ is proportional to $\sqrt{M}$. Hence, the threshold for $IPC_k(F, M)$ is k times the threshold for $IPC_1(F, M)$. The threshold fetch rate keeps proportional to the amount of ILP.

3.6 Consequences on the Fetch Engine Tuning

The threshold fetch rate should be evaluated according to the targeted processor and applications. If the threshold is below a single basic block per cycle, a simple fetch engine is enough. Otherwise, a more sophisticated fetch engine might be considered.

Increasing the fetch rate by allowing multiple branches to be predicted in a cycle is likely to impair branch prediction accuracy. This loss is acceptable when the effective fetch rate is close to the threshold. However, the performance of applications exhibiting a lower threshold value may be impaired.

Furthermore, a high fetch rate is not required when branches are often mispredicted. Indeed, both the instruction-cache and BTB misses lower the effective fetch rate. *Aliasing* on 2-bit counters and misses on the *Target Cache* lower branch prediction accuracy. Since clusters of capacity misses and clusters of capacity aliasing tend to occur simultaneously, all tables, i.e. instruction cache, BTB, and branch predictors, should be dimensioned proportionately.

Oversizing branch predictors brings little help when the application does not fit in the instruction cache, since increasing the prediction accuracy brings little gain when the fetch rate is low (see Figure 10).

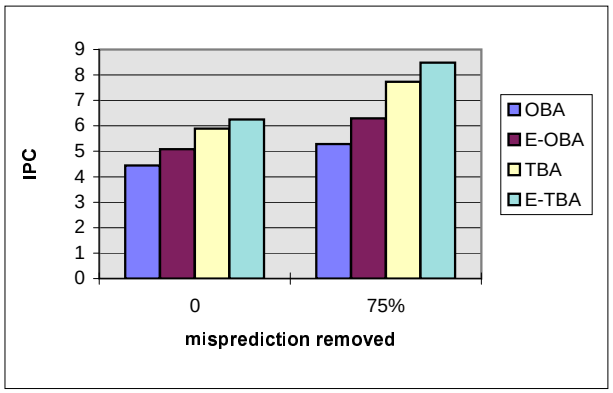


Figure 12: **Average IPC when removing 75 % of mispredictions**

3.7 Experimental Verification

The $\sqrt{N}$ model has shown that increasing the distance between mispredicted branch four-fold should double the threshold fetch rate. To emulate a better branch predictor than in simulations of Section 2, we randomly removed branch mispredictions. Simulation parameters remain the same as in Section 2, except for:

- the instruction window size is increased to 1k instructions
- the decode, rename, dispatch and issue widths are all enlarged to 32 instructions

Figure 12 shows the average IPC when 75% of mispredictions are removed. 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 between the two graphs whereas E-TBA speedup tops at 1.35. We verify that the new IPC for E-TBA is approximately twice the IPC for OBA with 4 times less mispredictions: to double the performance, we must double the fetch rate and reduce four-fold the number of mispredictions.

4 New Techniques Require Higher Instruction Fetch Bandwidth

The simulator used in the previous sections models a conventional out-of-order superscalar processor. In such a processor, instructions wait for all operands to be generated before being scheduled. Furthermore, all instructions belonging to mispredicted paths are discarded once the faulting branches are resolved. The performance is bounded by both data dependencies and branch prediction accuracy.

Some techniques have been recently introduced to overcome these limitations. As shown with the relations introduced in Section 3, these techniques require high-bandwidth fetch engines such as the E-TBA scheme or the trace cache.

4.1 Techniques to Alleviate Branch Mispredictions

Proposed hardware branch predictors are more and more complex to improve prediction accuracy. However, most recent branch prediction enhancements deal with space-efficiency. The accuracy of such predictors seems close to reach a limit [5].

The branch misprediction problem can be solved by other means. Such schemes are either hardware of software based, and they mimic the effect of increasing the branch prediction accuracy. By increasing the distance between mispredicted branches, these schemes require higher fetch bandwidth as shown in the previous section.

Predication. Predication is a software approach that extends conventional Instruction Set Architectures (ISA) to turn control dependencies into data dependencies. Hence, a lower number of branch instructions. A recent example is the new HP/Intel IA-64 ISA [12]. Predication allows to significantly increase the distance between mispredicted branches [18, 24]. [4] showed that when predication is applied to well chosen branches, the benefit of a larger pool of valid instructions overcomes the addition of new data dependencies. Predicated binaries are likely to require higher fetch bandwidth because of both the increased distance between mispredictions and the instruction fetching overhead. Longer basic blocks should also benefit the fetch engines introduced in Section 2. Finally, note that predicates may be dynamically predicted.

Selective Eager Execution. Instead of predicting the outcome of a branch, execution may process instructions from both paths. Eager execution is a well known alternative to branch prediction [25], which mimics a perfect branch predictor. Since tens of millions of transistors are currently available on-chip and since big branch prediction accuracy improvements are unlikely, some form of eager execution may become viable. Promising solutions [13, 16, 1] use a confidence estimator [14] to isolate difficult-to-predict branches, and to fork to both paths only on these branches. A high instruction fetch bandwidth is needed because of both the higher threshold fetch rate and the instruction fetching overhead due to processing both paths simultaneously. The fetch engine has been reported to be a performance bottleneck in [16] and [13] where eager execution was studied. [1] showed that performance keeps increasing up to 32 instructions fetched in a single cycle.

4.2 Techniques to Alleviate Data Dependencies

Some recently proposed techniques have been effective in reducing the negative performance impact due to data dependencies. By cutting data dependencies, these technique increase the amount of parallelism that can be extracted from fixed-size instruction window. As the

threshold fetch rate increases linearly with the available parallelism, such techniques require higher instruction-fetch bandwidth.

Data value prediction. Data value prediction has been proposed in [17] to exceed the dataflow limit. Instructions tend to produce the same value repeatedly [17], or some successive values that differ by a constant delta [10]. Such patterns are highly predictable. When predictions are correct, this scheme removes edges in the data-flow graph. Hence, shortening critical paths. [9] emphasizes the requirement in high fetch bandwidth to fully exploit the potential of value prediction.

Data dependence collapsing. Chains of dependent instructions can be collapsed and executed as one operation [27]. For instance, two dependent ADD instructions can be executed in a single cycle using an adder operating on three source operands [19]. Like value prediction, this scheme removes data dependencies.

Instruction reuse. Instruction reuse [29] tracks operand values in order to reuse past results. The tracking is done either by detecting operand value changes or by performing some value comparisons. Successful result reuses are completely safe, and they lead to both the elimination of the instructions and the removing of data dependencies.

Speculative memory bypassing. As the instruction window grows, memory dependencies (collisions) become more and more common. These dependencies can be predicted [21], and translated into register dependencies by the *speculative memory bypassing* scheme [22, 15]. [22] showed that *speculative memory bypassing* increases substantially the amount of ILP.

5 Summary and Concluding Remarks

In this paper, we emphasized the relations between effective fetch rate, available parallelism, and branch prediction accuracy.

First, we have introduced new enhancements to boost instruction-fetch bandwidth of conventional fetch engines. By allowing the bypassing of one not-taken branch in the Two-Block Ahead scheme, E-TBA gets 20 % extra instruction fetch bandwidth at a modest hardware overcost. This extra instruction fetch was found not be really useful on superscalar processor executing on a single path and relying on true data dependency. However, new techniques such as eager execution and data value prediction will consume this bandwidth.

The second contribution of this paper is the analysis of the relations between available parallelism, branch prediction accuracy and instruction fetch rate. We experimentally confirmed that the available parallelism in an instruction window grows approximately as the square root of its size. That is, the instruction parallelism extractable from a program grows as the square root of the inverse of the misprediction rate. We analytically shown that

there is very poor performance benefit of increasing the I-fetch rate over a threshold also proportional to the square root of the distance (in instructions) between two consecutive mispredictions.

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, let us say instruction fetch, some other parameters cannot be completely explored. For example, it would not have been possible for us to simulate one after another all the mechanisms presented in Section 4. Hence, all simulation studies explored only a small subset of the possible design space. Simulating "true" mechanisms, but not varying their parameters, leads to biased simulation results. A typical example is represented by our own original experiment in Section 2.5 where 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 have been the other way around.

This leads us to argue for deemphasizing on complete processor simulation, but rather to focus on more significant metrics than just the overall simulated performance of the processor. For instance, the future need for higher instruction fetch bandwidth is clearly identified in Section 4. Then, the average fetch rate and the branch misprediction rate are better metrics to discriminate between two instruction fetch engines such as TBA and E-TBA than the simulated performance of a fixed execution core.

References

[1] Pritpal S. Ahuja, Kevin Skadron, Margaret Martonosi, and Douglas W. Clark. Multipath execution: Opportunities and limits. In *Proceedings of the 12th ACM International Conference on Supercomputing*, 1998.

[2] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support fro Programming Languages and Operating Systems*, 1994.

[3] 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.

[4] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[5] I.-C.K. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[6] 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.

[7] 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.

[8] D.H. Friendly, S.J. Patel, and Y.N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the the 30th Annual International Symposium on Microarchitecture*, December 1997.

[9] Freddy Gabbay and Avi Mendelson. The effect of instruction fetch bandwidth on value prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[10] Freddy Gabbay and Avi Mendelson. Using value prediction to increase the power of speculative execution. *ACM Transactions on Computer Systems*, 16(3), August 1998.

[11] Linley Gwennap. Digital 21264 sets new standards. *Microprocessor Report*, 10(14):11–16, October 1996.

[12] Linley Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14), October 1997.

[13] T.H. Heil and J.E. Smith. Selective dual path exeution. Technical report, University of Wisconsin, Madison, November 1996. http://www.ece.wisc.edu/~jes/.

[14] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictors. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[15] 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 31th Annual International Symposium on Microarchitecture*, 1998.

[16] 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.

[17] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit with value prediction. In *Proceedings of the 29th International Symposium on Microachitecture*, 1996.

[18] 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.

[19] N. Malik, R.J. Eickemeyer, and S. Vassiliadis. Interlock collapsing ALU for increased instruction-level parallelism. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[20] 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.

[21] A. Moshovos, S.E. Breach, T.N. Vijaykumar, and G.S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[22] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[23] Sanjay Patel, Marius Evers, and Yale Patt. Improving trace cache effectiveness with branch promotion and trace packing. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[24] Dionisios N. Pnevmatikatos and Gurindar S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, 1994.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] A. Sodani and G.S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[30] 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.

[31] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[32] S. Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Proceedings of the 3rd symposium on High Performance Computer Architecture*, February 1997.

[33] 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.

[34] 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.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399