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

Pierre Michaud
IRISA/INRIA
pmichaud@irisa.fr

André Seznec *
Compaq Computer Corporation
seznec@vssad.hlo.dec.com

Stéphan Jourdan
Intel Corporation
sjourdan@ichips.intel.com

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 fetch 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 fetch 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 of size N grows as the square root $\sqrt{N}$, 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.

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 [15] and the more conventional multiple-block fetching approach [21, 4, 17]

Relations between specific parameters are seldom analyzed in published studies. For instance, this is the case between the instruction-fetch rate, the branch prediction accuracy, and the available parallelism. 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 per 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. Section 4 provides some concluding remarks.

Experimental Set-Up. All experiments in this study are conducted on the IBS traces [19]. 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 branch predictors [21, 4, 17]. The other option is based on a *trace cache* [15].

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:

*This work was done while the second author was with IRISA/INRIA

- 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 [17] 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 DEC 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 [4]. Instead of fetching basic blocks, we would fetch sequential traces, a sequential trace ending on the first taken branch.

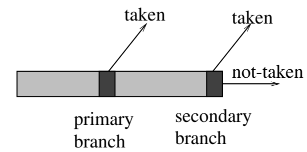

Figure 1. An extended block: when the primary branch is not-taken, the block is extended up to the secondary branch

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. 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* [22]: 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.

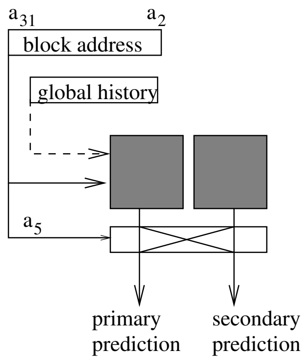

Figure 2. E-OBA scheme: the secondary prediction is used if the primary branch is not-taken

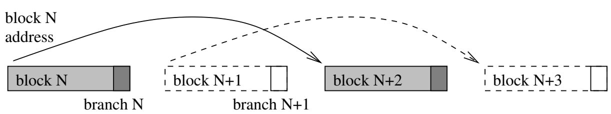

Figure 3. Principle of TBA prediction

2.3. The Basic TBA Scheme [17]

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 [17].

As described in [17] , 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 [17], 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.

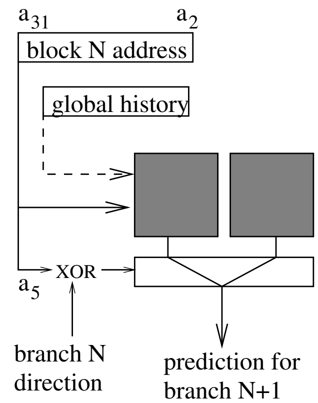

Figure 4. TBA indexing on a 2-bit counter table, a BTB, or a target cache

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 : 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

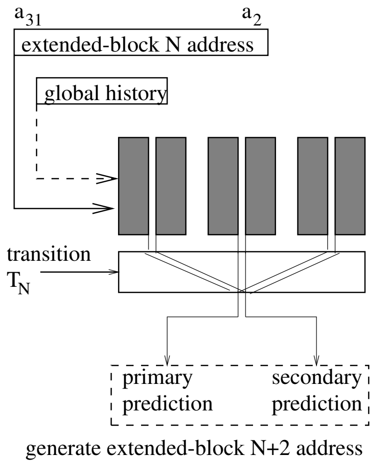

Figure 5. E-TBA scheme (logical)

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 experimental results in this study are obtained through trace-driven simulations using the IBS traces.

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-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* [12] 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 [17] 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 6 and 7 show respectively IPC values (instructions retired per cycle) and fetch rates (valid instructions fetched per fetch cycle on the right path). 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 (we do not count instructions fetched after a mispredicted branch). 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. On average, there are approximately 250 instructions per mispredicted branch (more precisely, taking into account exceptions and interruptions in the IBS traces, there is an average of 200 instructions per control flow break).

As expected, higher fetch rates always improve performance (Figure 6). 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. 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

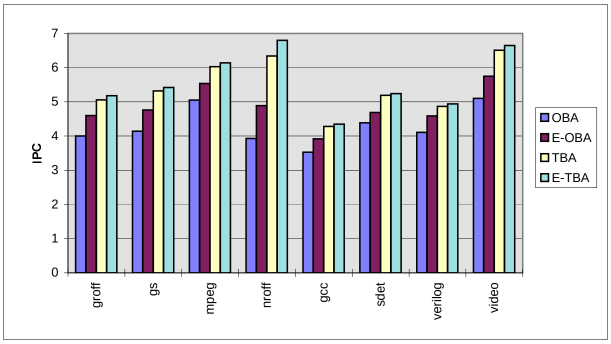

Figure 6. Measured IPC

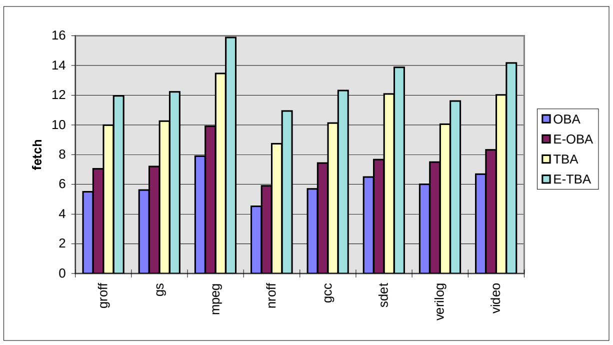

Figure 7. Average fetch

Results are reported in Figure 8, measured in cycles per instruction (CPI), and averaged over all the benchmarks but *sdet* (which consists mainly in a single tight loop, with a lot of full-buffer cycles). 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, could 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 schemes become even more effective when taking into consideration compiler techniques to increase the proportion of not-taken branches, as in [1].

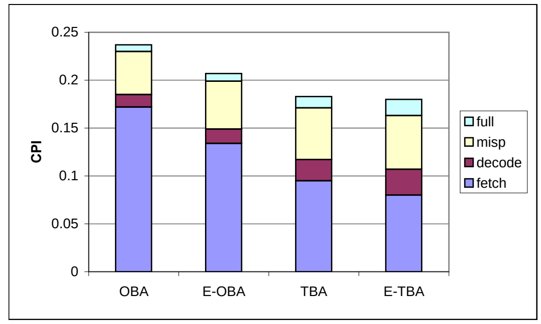

Figure 8. CPI breakdown (average)

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. We give further explanation on how superscalar processors can take better advantage of the extra instruction fetch bandwidth delivered by TBA and E-TBA.

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 [14, 20]. 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

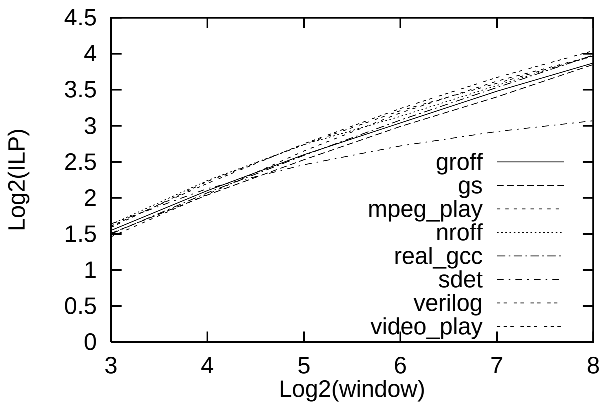

Figure 9. Average ILP as a fonction of the instruction window size (logarithmic scale)

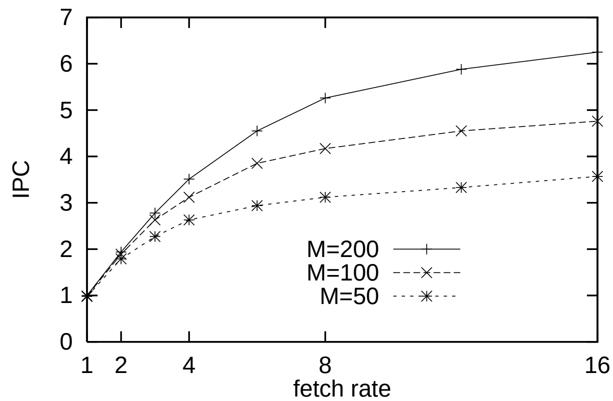

Figure 10. IPC in function of the fetch rate for M= 50,100 and 200 instructions on a $\sqrt{N}$ model

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 single tight 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 [14].

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 the behavior of a real system, such a model provides some insight by isolating important parameters.

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 the instructions before 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:

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

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 expose a threshold effect on the fetch rate: below the threshold, the performance increases quasi-linearly with the instruction fetch rate, whereas above the threshold, the performance is close to a limit.

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 better branch prediction accuracies. The diminishing performance return of increasing the fetch rate, observed in Section 2.5, 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 $ILP = \sqrt{N_i}$ in each cycle, assuming M=200 instructions, and for different fetch rates (the area below one curve is equal to M instructions). 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, in Figure 11). Beyond the threshold, increasing

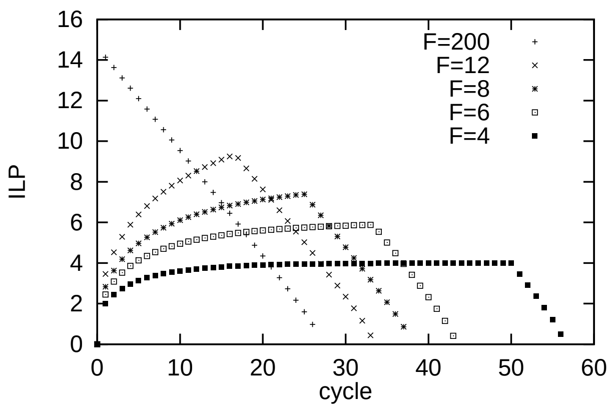

Figure 11. Dynamic ILP $= \sqrt{N_i}$ for M=200

the fetch rate allows some instructions to execute sooner, but this barely affect the total execution time: these instructions are not on the critical path.

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 that the slope of the falling ILP phase is nearly constant (Figure 11):

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

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 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 in Figure 10 are homothetic. One conclusion of this is, to double the performance, we should increase the distance between mispredicted branches fourfold 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 overcome 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

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

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.

Moreover, a high branch prediction accuracy requires a high fetch rate. Indeed, BTB misses lower the effective fetch rate (valid instructions decoded per cycle), while aliasing on 2-bit counters lower the branch prediction accuracy. Since clusters of capacity misses and clusters of capacity aliasing tend to occur simultaneously, it is worth removing capacity aliasing only if we remove capacity BTB misses at the same time: the BTB and other predictor tables should be dimensioned proportionately.

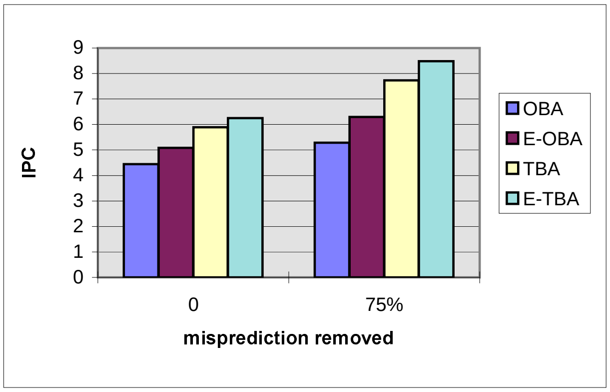

Figure 12. Average IPC when removing 75 % of mispredictions

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 2.5, we randomly removed branch mispredictions. The processor configuration is almost the same as in Section 2.5, except that to scale with better branch prediction, we increase the processor instruction window from 256 to 1024, and the pipeline width from 16 to 32.

Figure 12 shows the average IPC on the new processor configuration, when no mispredictions are removed and when 75% of mispredictions are removed respectively. 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: to double the performance, we must double the fetch rate and reduce four-fold the number of mispredicted branches.

3.8. New Techniques Require Higher Instruction Fetch Bandwidth

As shown in this section, the instruction-fetch bandwidth requirement depends on the branch prediction accuracy and the ILP. The processor we simulated in Section 2.5 had a limited branch prediction accuracy, and was constrained by true data dependencies. The use of techniques to overcome these limitations will increase the fetch bandwidth requirement.

Techniques to Alleviate Branch Mispredictions Dynamic branch predictors are more and more complex in order to improve prediction accuracy. However, most of the recent branch prediction enhancements deal with space-efficiency. The accuracy of such predictors seems close to reach a limit [3]. 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 a higher fetch bandwidth (measured in **useful** instructions per cycle). Techniques to increase the distance between mispredicted branches include *predication* (both static [11] and dynamic [8]) and *selective eager execution* [9].

Techniques to Alleviate Data Dependencies Some recently proposed techniques have been effective in reducing the negative performance impact of data dependencies. By cutting data dependencies, these techniques increase the amount of parallelism that can be extracted from a fixed-size instruction window. As the threshold fetch rate increases linearly with the available parallelism, such techniques require a higher instruction-fetch bandwidth. These techniques include *data value prediction* [10], *data dependence collapsing* [16], *instruction reuse* [18] and *speculative memory bypassing* [13, 7] It should be noted that the requirement for a higher instruction-fetch bandwidth, when using data value prediction, has been emphasized in [6].

4. Summary and Concluding Remarks

In this paper, we emphasized the relations between the instruction-fetch rate, 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. 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.

The second contribution of this paper is an analysis of the relations between available ILP, branch prediction accuracy and instruction-fetch rate. We observed that, on average, the available parallelism in an instruction window grows roughly as the square root of its size, and we derived an empirical model from this observation.

This model 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 rate and ILP grow as the square root of the distance (in instructions) between two consecutive mispredicted branches. We experimentally confirmed that, to double the performance, we should both double the fetch rate and decrease four-fold the number of mispredicted branches.

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. For example, it would not have been reasonable for us to simulate one after another all the mechanisms referenced in Section 3.8. Hence, our study in Section 2.5 explored only a small subset of the possible design space.

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.

References

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

[2] P.-Y. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

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

[4] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[5] K. Driesen and U. Holzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, 1998.

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

[7] 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.

[8] 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.

[9] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

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

[11] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. mei W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[12] 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.

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

[14] E. Riseman and C. Foster. The inhibition of potential parallelism by conditional jumps. *IEEE Transactions on Computer Architectures*, C-21(12):1405–1411, Dec. 1972.

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

[16] Y. Sazeides, S. Vassiliadis, and J. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th International Symposium on Microarchitecture*, 1996.

[17] 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.

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

[19] 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.

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

[21] T.-Y. Yeh, D. Marr, and Y. 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.

[22] T.-Y. Yeh and Y. Patt. Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors. In *Proceedings of the 26th International Symposium on Microarchitecture*, 1993.