Multiple-Block Ahead Branch Predictors

to appear in Proceedings of ASPLOS VII, Boston, October 1996

André Seznec † Stéphan Jourdan ‡ Pascal Sainrat ‡ Pierre Michaud †*

IRISA †
Campus de Beaulieu
35042 Rennes, France
{seznec,pmichaud}@irisa.fr

IRIT ‡
Université Paul Sabatier
31062 Toulouse, France
{jourdan,sainrat}@irit.fr

Abstract

A basic rule in computer architecture is that a processor cannot execute an application faster than it fetches its instructions. This paper presents a novel cost-effective mechanism called the two-block ahead branch predictor. Information from the current instruction block is not used for predicting the address of the next instruction block, but rather for predicting the block following the next instruction block.

This approach overcomes the instruction fetch bottleneck exhibited by wide-dispatch "brainiac" processors by enabling them to efficiently predict addresses of two instruction blocks in a single cycle. Furthermore, pipelining the branch prediction process can also be done by means of our predictor for "speed demon" processors to achieve higher clock rate or to improve the prediction accuracy by means of bigger prediction structures.

Moreover, and unlike the previously-proposed multiple predictor schemes, multiple-block ahead branch predictors can use any of the branch prediction schemes to perform the very accurate predictions required to achieve high-performance on superscalar processors.

1 Introduction

Two different approaches are used in current processors to achieve high performance: "brainiacs vs. speed demons" [6]. While "brainiacs" favor the parallel execution of instructions and "speed-demons" favor a high clock rate, both approaches are facing a similar difficulty with fetching instructions at a sufficient rate. The purpose of this paper is to propose a new branch prediction mechanism allowing to increase the instruction fetch rate for both approaches.

"Brainiac" processors To best exploit the available ILP, "brainiac" processors are using a large number of functional units working in parallel.

Unfortunately, the instruction-fetch mechanisms implemented in current commercial microprocessors do not fully exploit the potential parallelism. For these processors, the instructions fetched in a single cycle most often belong to the same basic block, and are not usually permitted to span two cache lines. Since a processor cannot execute instructions faster than it fetches them, these constraints significantly impair performance, particularly on codes featuring many small basic blocks.

A partial solution to the instruction fetch bottleneck, is to fetch instructions belonging to multiple consecutive basic blocks, as is done in processors such as the POWER2 [18]. To solve the whole problem, multiple non-consecutive basic blocks must be fetched in a single cycle as most basic blocks are only five instructions long. Indeed, the potential parallelism has been shown to be higher than six instructions per cycle in general-purpose integer applications while assuming a perfect instruction-fetch mechanism [11]. A processor featuring such a mechanism would have to predict multiple targets and branch outcomes in a single cycle.

In superscalar processors, blocks of consecutive instructions are fetched in parallel. The last instruction of such a block is either a branch or is determined by some implementation constraints (for instance, the boundary of a cache block or the maximum number of instructions in the block). Throughout this paper, we refer to processors that can fetch only one basic block per cycle as *single I-fetch processors*, to processors that can fetch two non-consecutive blocks per cycle as *double I-fetch processors*, and to *multiple I-fetch processors* as an extension to the latter case.

We show in this paper that double I-fetch processors will have a major performance advantage over single I-fetch processors for a dispatch width of six or higher. Our belief is that future generations of "brainiac" processors will be multiple I-fetch processors.

"Speed-demon" processors To achieve high performance, "speed demon" processors rely on moderate numbers of functional units, but a very high clock rate. In such processors, a single instruction block is dispatched in each cycle, but the branch predictor is often a critical path in the processor. In current microproces-

*This work was partially supported by PRC-GDR AMN (CNRS)

sors, either the branch prediction and the address generation are completed in a single cycle, or pipeline bubbles are inserted on each predicted taken branch (e.g. on DEC 21164 [5], PentiumPro [8] or MIPS R10000 [13]), therefore potentially limiting the performance. Another way to deal with this critical path is to reduce the number of entries of the one-cycle access prediction structures (e.g. HP PA-8000 [7]), thus impairing the predition accuracy.

In this paper, we will show that pipelining the branch prediction is possible on single I-fetch processors, and that branch prediction therefore need not be a critical path on such processors.

The two-block ahead branch predictor In conventional branch prediction mechanisms, information associated with the current instruction block such as its memory address, is used to predict the next instruction block. Previous multiple predictors [19, 4] also rely on a single piece information to predict the two subsequent instruction blocks.

In this paper, we propose a complete and cost-effective mechanism called the *Two-Block Ahead Branch Predictor*. The originality of our mechanism is to use information associated with the current instruction block to predict the block following the next instruction block. Such an approach can obviously be extended to predict blocks with even further advance; we refer to this as a *Multiple-Block Ahead Branch Predictor*.

The *Two-Block Ahead Branch Prediction* consists of a two-block ahead branch prediction table, a two-block ahead branch target buffer and a two-block ahead return stack.

Most previously proposed prediction schemes can be adapted to the *Two-Block Ahead Branch Predictor* leading to high prediction accuracy. Moreover, the amount of information stored in the two-block ahead branch predictor is not higher than in a conventional branch prediction mechanism.

The two-block ahead predictor can be used in a double I-fetch processor: both fetch addresses are used to predict the two subsequent instruction blocks to fetch on the next cycle. Implementing the two-block ahead branch predictor in a single I-fetch “speed demon” processor allows the branch prediction to be pipelined.

Paper organization The remainder of the paper is organized as follows. Related work is discussed in section 2. Section 3 compares single I-fetch and multiple I-fetch processors. Section 4 introduces the two-block ahead branch predictor and presents its implementation for a double I-fetch processor. Section 5 shows that the branch prediction process can be pipelined in single I-fetch “speed demon” processors by means of our two-block ahead branch predictor. Finally, simulation results are reported in section 6. Section 7 concludes the paper.

2 Related Work

To our knowledge the pipelining of the branch prediction has never previously been addressed.

Only a few studies [19, 4, 3] have addressed the problem of fetching multiple non-consecutive basic blocks in a single cycle.

Yeh, Marr, and Patt To fetch two basic blocks in a single cycle, Yeh et al. [19] proposed storing 6 addresses in each entry of their *Branch Address Cache* (BAC): T, N, TT, TN, NT, and NN, where N and T refer to the outcome of the branches (*not-taken* and *taken* respectively). The branch prediction mechanism can predict two branches in a single cycle. According to the prediction made by a history-only based scheme (address-based schemes give lower prediction accuracy since they use the same address for both predictions), the addresses of the two subsequent basic blocks are returned with a hit in the BAC. When a branch is resolved for the first time, an entry is allocated in the BAC with fields T and N set (primary fields). If the previous fetch address had a valid primary branch entry in the BAC with secondary fields cleared, and if there was enough bandwidth to fetch another basic block, then T and N are also inserted in these secondary fields. This introduces wasted fields when entries are allocated for primary branches with not enough fetch bandwidth left for a second basic block (with a dual-ported instruction cache, this occurs each time a basic block lies across an aligned block boundary). Furthermore, since most of the branches are unidirectional, two third of the fields are under-utilized.Finally, a basic block can belong to several BAC entries, so their mechanism does not require any static partitioning. Redundancies are however created, but the scheme does not rely on any compiler work.

Dutta and Franklin In [14], the authors proposed splitting the *Control Flow Graph* (CFG) into subgraphs. To fetch two non-consecutive basic blocks even when they belong to different cache lines, they use tree-like subgraphs of depth 3 [4]. Nodes of the subgraphs are straightline pieces of code (basic blocks). Intermediate nodes (depth 0 and 1) can be terminated by any control-changing instructions while the last nodes (depth 2) are terminated by single-target instructions (either unconditional branches, procedure calls, or non-branch instructions), and their lengths are limited by the instruction-fetch bandwidth. Intermediate outcomes are not predicted. Instead, one path is predicted in the subgraph among four. All parameters required to describe a subgraph are stored in a Subgraph History Table (SHT). Except for the prediction mechanism, this method is much like Yeh's approach. To avoid redundant information in the SHT, each basic block should belong to only one subgraph. Their scheme mostly relies on compiler work to partition the CFG into tree-like subgraphs of depth 3. One should note that basic block duplication implies that history information is now shared out. Hence, redundancy impairs more significantly performance than in Yeh's approach. Furthermore, since each entry in the SHT holds a rigid subgraph structure, there might be many under-utilized or wasted fields. Indeed, a static CFG is not as simple as a tree and it cannot be perfectly partitioned into tree-like structures. With optimized code, the prediction mechanism gives good accuracy results compared to non-hybrid schemes without a lot of additional logic.

Conte, Menezes, Mills, and Patel In [3], the authors introduced a mechanism called the *Collapsing Buffer* that achieved *merging* [10]. This mechanism can fetch multiple basic blocks in a single cycle as long as they belong to the same cache line, and otherwise performs some alignments between two basic blocks by means of a pipelined fetch mechanism (*banked sequential*). The mean instruction-fetch throughput is at most one cache line with no restriction on the number of predicted branches within the cache line. The scheme features an interleaved coupled BTB/BHT providing one entry to each instruction of a cache line. The Collapsing Buffer scheme is efficient as long as branch targets address the same cache line and performs well on their execution model retiring less than 2.5 instructions per cycle on integer applications. Another major drawback is the requisite use of an address-only based prediction scheme. Moreover, as the I-cache line size keeps growing in current processors, the interleaving factor of the BTB grows as well and the collapsing logic becomes more complex. Although this approach is interesting, our purpose is to go one step beyond by fetching multiple basic blocks in a single cycle, even when they belong to different cache lines. One should note that our approach is not incompatible with collapsing.

Generic comment The first two methods create several layers of BTB: BTB1, BTB2 where BTB1 makes a prediction one branch ahead and BTB2 makes a prediction two branches ahead (it happens that both combine BTB1 and BTB2 into the same structure, *joining* on the search key). Our elegant approach only uses a single kind of BTB, BTB2, and dual-ports it to achieve two predictions and address computations in a single cycle. Hence, there is no difference between the branch prediction numbers in terms of the first and second prediction.

It should be noted that none of these previously published solutions can be extended easily to pipeline the branch prediction.

3 Single vs. Multiple I-fetch Processors

While multiple basic block fetch mechanisms have already been proposed, there was no clear study showing that such mechanisms would provide performance enhancements. The purpose of this section is to show that despite data dependencies and resources hazards, multiple fetch mechanisms would give significant performance improvement in wide-dispatch out-of-order processors.

Traces The experimental results presented in this paper are based on the programs from the SPEC92 suite [17]. Programs from both the CINT92 and the CFP92 collections are considered in the presented evaluations. The benchmarks were compiled on a R4600-based SGI workstation using cc and the standard makefiles provided with the suite (with all optimizations turned on). We used the PIXIE profiler [16] to collect instruction traces from a real processing of the SPEC benchmarks, including library calls. These traces fed our simulator.

Due to time constraints, the smallest input files or slightly modified versions have been used in order to run the programs to completion. In all, more than 600 million instructions have been captured (with all NOPs removed from the traces).

Machine Model The modeled architecture depicted in figure 1, implements out-of-order and speculative execution policies in order to best exploit ILP. In brief, after being fetched, instructions are decoded and dispatched in-order from the instruction-dispatch buffer to

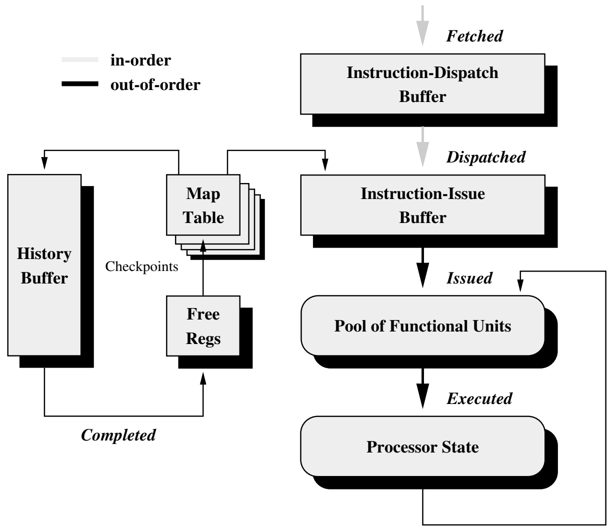


Figure 1: Simulated Out-of-Order Microarchitecture.

the instruction-issue buffer. The upper bound of the number of instructions dispatched each cycle defines the dispatch width. Registers are renamed using a map table during the dispatch process. Instructions in the issue buffer may be issued out-of-order when all their operands are available, and a *max-dependent* selection mechanism as described in [1] is used when more than one instruction compete for the same functional unit access. To enforce precise interrupt management, a history buffer similar to the active list of the MIPS R10000, records the previous mappings discarded by the renaming process during the dispatch stage. Checkpoints of the map table (architectural state) are established at every branch in order to recover from branch misprediction in one cycle, regardless of the number of mapping modifications recorded in the history buffer. With such a scheme, the history buffer is only used to recover from other exceptions and to keep track of the state of the physical registers in order to free them when the instructions complete. When an instruction finishes execution, its result updates the processor state but its corresponding entry remains in the history buffer until all previously dispatched instructions can no longer produce interrupts. Instructions capable of generating interrupts are conditional and indirect branches (mispredictions), divides, and memory accesses. In the latter instruction class, subsequent entries may be dequeued as soon as the address is computed. Each cycle, multiple out-of-order instruction retirements can be made, freeing the physical registers to be reused in the renaming process.

A previous study [11] has shown that configurations reported in table 1 of such out-of-order architectures give almost no performance loss over perfect configurations only limited by the size of the lookahead window, assuming an ideal-fetch mechanism and no misprediction. The instruction latencies used in the simulations were those of the PowerPC 604 [9]. The mean-IPC values varied from 3.6 to 6.5 on integer programs according to the dispatch width (4, 6, and 8 instructions dispatched per cycle). We keep their configurations (DW 4, DW 6 and DW 8) in order to evaluate the fetch mechanisms. The lookahead window is the maximum number

Parameters	DW4	DW6	DW8
Dispatch Width	4	6	8
Lookahead Window Size	32	64	96
Issue Buffer Depth	28	48	72
Fixed-Point Units	3	4	5
Floating-Point Units	2	2	2
Branch Units	2	2	3
Data-Cache Ports	2	3	4

Table 1: Model Configurations.

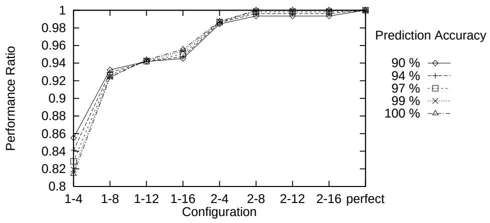
of dispatched instructions that can be processed at the same time, sometimes referred as the instruction window. Moreover, we assumed for all the models a unified issue-buffer, and a maximum number of 16 checkpoints. Such a value does not degrade the performance of any of the models.

In most processors, the fetch mechanism consists mainly of three parts: an instruction cache from where the instructions are fetched, an instruction-dispatch buffer where the instructions are maintained waiting to be dispatched, and some branch prediction structures predicting the outcome and the target address of any fetched branch. The dispatch buffer decouples the instruction fetching from the dispatch process, sustaining a better throughput in the presence of cycles in which only a small number of instructions can be fetched (the buffer can be filled in a single cycle). Perfect instruction and data caches are used throughout this section.

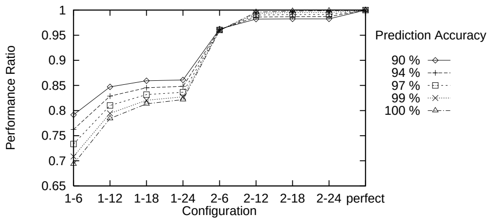
Multiple I-fetch is useless with long basic blocks The CFP92 subset features long basic blocks close to fifteen instructions long on average. Simulations on floating-point programs, not reported here, have shown that a single I-fetch processor is effective whatever the dispatch width may be provided a deep dispatch buffer. A 8-wide single I-fetch processor gives over 97.3 % of the perfect performance when the prediction accuracy is higher than 90 %.

Double I-fetch is useful with small basic blocks The average size of the basic blocks in the CINT92 subset is five instructions long. Figures 2 (a) through (d) show the results on integer programs according to the depth of the instruction-dispatch buffer, the prediction accuracy, and the dispatch width. Only the geometric means of the results are reported. Configurations are based both on the number of basic blocks fetched in a single cycle, and on the depth of the dispatch buffer. For instance, configuration 2-8 denotes a double I-fetch processor featuring a 8-deep dispatch buffer.

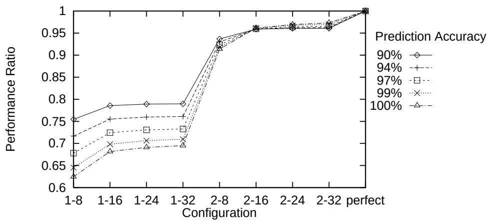
In (a), (b), and (c), performance ratio is the relative performance (IPC) between the evaluated configuration and a configuration featuring a perfect fetch mechanism, the prediction accuracy remaining the same. The curves clearly state that 4-wide processors do not require any improvement over a single I-fetch policy except for a 8-deep dispatch buffer, giving 93 % of the perfect performance whatever the prediction accuracy may be. On the other hand, 6-wide and especially 8-wide double I-fetch processors give a huge improvement of performance over single I-fetch processors. In a 8-wide processor, the improvement is between 20 and 40 % depending on the prediction accuracy, assuming a deep



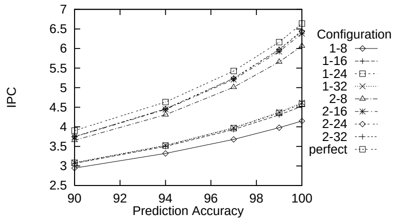
(a) Dispatch Width: 4



(b) Dispatch Width: 6



(c) Dispatch Width: 8



(d) Dispatch Width: 8

Figure 2: Performance of Single and Multiple I-Fetch Processors.

dispatch buffer. Moreover these results bring to light that fetching more than two basic blocks in a single cycle is not effective for an 8-wide machine (while keeping binary compatibility) as double I-fetch mechanisms provide nearly 100 % of performance. One should note that the relative benefit when fetching two basic blocks in a single cycle increases with the branch prediction accuracy. Finally, an instruction buffer twice as big as the dispatch width is required in any case.

As shown in figure (d) for a 8-wide processor, it is more effective to increase the number of blocks fetched per cycle than to improve the prediction accuracy. Nevertheless, these two optimizations are not exclusive and they each give new opportunities for performance improvement.

4 The Two-Block Ahead Branch Predictor

This section is illustrated with the implementation of a Two-Block Ahead Branch Predictor for a double I-fetch processor (figure 4). The implementation of a pipelined Two-Block Ahead Branch Predictor for a single I-fetch processor will be detailed in the next section.

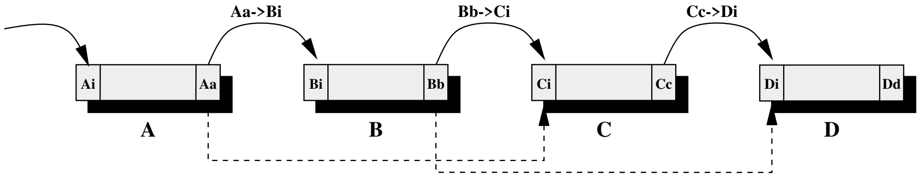
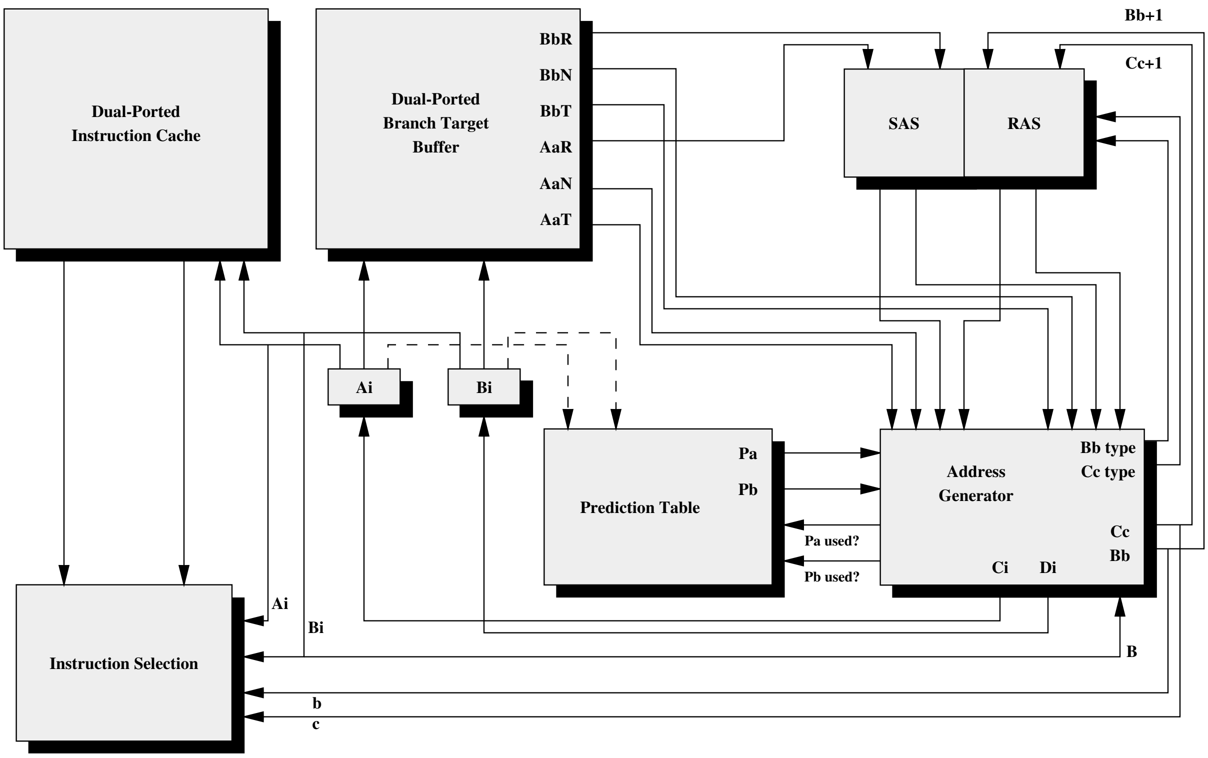
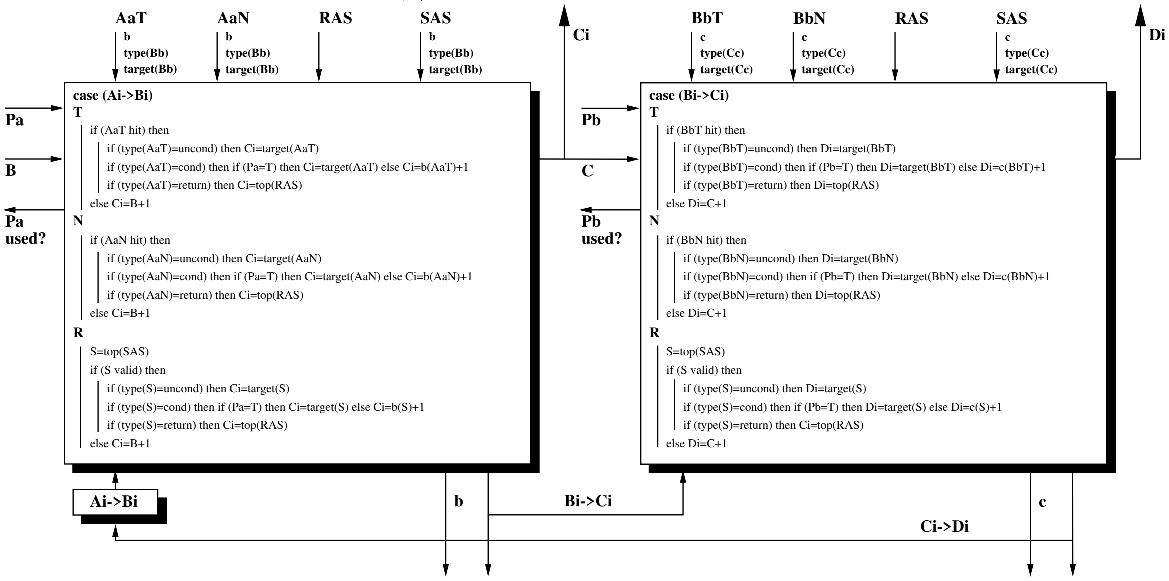


Figure 3: Two-Block Ahead Branch Information.



(a) Predicting Two Basic Blocks



(b) Instruction Address Generation

Figure 4: Two-Block Ahead Branch Predictor in Double I-Fetch Processor.

As stated in the introduction, the two-block ahead branch predictor uses information associated with the current instruction block to predict the address of the instruction block that is two blocks ahead. Its principle is illustrated in Figure 3 where **Ai**, **Bi**, **Ci**, and **Di** are the basic block starting addresses and **Aa**, **Bb**, **Cc**, **Dd** are the branch addresses. Any of the control-flow transitions can be fall-through. While the instruction blocks **A** and **B** are fetched, the two-block ahead branch predictor uses **Aa** instead of **Bb** to predict **Ci**, and **Bb** to predict **Di**. The behavior of each part of the predictor is described in the following sections.

4.1 The Two-Block Ahead Branch Prediction Table

Instead of using the address and the history register of the conditional branch (**Bb**,**Hb**) to predict its outcome **Ci**, our scheme always uses the address and the history register of the previous branch (**Aa**,**Ha**) to predict **Ci**. Such a scheme can be adapted to use any branch prediction schemes combining address and history (see for instance [12]).

We will show in the section 6 that the prediction is as accurate as if the address of the branch had been used instead. In figure 4, **Pa** and **Pb** refer to the predicted outcome when the Branch Prediction Table (PT) is indexed with (**Aa**,**Ha**) and (**Bb**,**Hb**) respectively.

4.2 The Two-Block Ahead Branch Target Buffer

BTB entry description The two-block ahead BTB records information for a given branch in an entry associated with both the address of the previously fetched block and the type of transition between both blocks.

When a taken branch **Bb** is mispredicted or misfetched, a BTB entry is allocated to record its target (**Ci**), type (conditional, unconditional, indirect, or return) i.e. 2 bits, and position **b** in block **B** (for instance, 4 bits for a 16-instruction cache-line size).

Unlike conventional designs, the BTB entry is not tagged with the address **Bb**, the starting address **Bi** of the instruction block containing **Bb**, or the address of the cache line containing **B**. But it is associated with the address of **Aa**, the last instruction in the previous instruction block. The BTB is indexed with (1) the address of **Aa**, and (2) the type of the transition between **Aa** and **Bi** (**Aa**→**Bi**) (2 bits). If no branch was fetched with **A**, **a** would be the last instruction in block **A**. There are three types of transition: **T** (**Aa** is a non-return taken branch), **N** (**Aa** is a non-taken conditional branch or a non-branch instruction), and **R** (**Aa** is a call). Type **R** is special and is further explained when we introduce the procedure return mechanism.

BTB read We illustrate here the read of the two-block ahead branch target buffer on a double I-fetch processor. Let us detail the information that is available at the beginning of the cycle.

- both addresses **Ai** and **Bi** are available; the block addresses **A** and **B** are used to access the BTB in addition to the I-cache.
- the branch position **a** and the transition type **X** between **Aa** and **Bi** were determined during the previous cycle.

All the information required to compute **Ci** is known. We can therefore check for BTB entry **AaX** to compute **Ci**. This entry would hold the target **Ci** of branch **Bi**, its type and its position. If **Bb** is a conditional branch, the outcome is provided by the two-block ahead branch prediction table. The whole process for computing **Ci** is detailed in figure 4.

Some pieces of information needed for computing **Di** are not directly available at the beginning of the cycle: position **b** of the branch in block **B** and transition type **Y** from **Bb** to **Ci**. This information is obtained on the fly with **Ci**:

- if **AaX** hits in the BTB then this information is part of the **AaX** entry.
- if **AaX** misses in the BTB then we assume that no branch in **B**, and **b** would be the last instruction in line **B** and transition **Y** is assumed to be fall-through.

Once these values are produced, the tag checking for the BTB entry **BbY** may begin and the address **Di** is computed in a similar way as **Ci**.

To enable parallel read of both entries **AaX** and **BbY**, any entry **EeZ** is mapped in the BTB as follows: low-order bits of **E** are used to address the set (BTB indexing), and both **eZ** and high-order bits of **E** are used to tag the entry allocated within the set.

So indexing the BTB with **A** and **B** may be done in parallel, but the tag-matching for **AaX** and **BbY** is partially serialized. This constraint is part of the parallelism vs. speed tradeoff ("brainiacs vs. speed demons"). However, such a process can be easily pipelined within the fetch stage, further featuring the structure update process.

Storage cost The amount of information stored in a two-block ahead BTB entry is only a few bits wider than in a conventional BTB (position **b** of the branch in **B** and the transition type **Aa**→**Bi**).

On the other hand, two entries may be associated with a single address **Aa**. The BTB entry **AaT** records information for a branch in the target basic block of branch **Aa**, and entry **AaN** records information for a branch in the fall-through basic block of branch **Aa**.

Some redundancies may be created when a branch **Bb** has more than one predecessor block: in this case, a target may be represented several times in the BTB. However, our simulations show that the two-block ahead BTB does not require many additional entries to achieve the same hit ratio as a conventional BTB.

Associativity Since most of the conditional branches are either mostly taken or mostly fall-through [20], the BTB will often record only one of **AaT** or **AaN**. Thus the associativity required in the two-block ahead BTB will not be much higher than in a conventionnal BTB.

4.3 Coping with Procedure Returns

Procedure-return jumps are a special case where using a BTB alone is inefficient: the target address may change very often. To cope with this difficulty, many recent processors implement a Return Address Stack into which return addresses are pushed when the calls are fetched. The two-block ahead branch predictor may also use a Return Address Stack for predicting return addresses.

Nevertheless, when using two-block ahead branch prediction, the address **Bb** of the return branch should be used to predict the instruction block **Di** following the return target block (figure 5.a). **Di** is intuitively more dependent on the return target **Ci** which may vary frequently for the same return branch than on the return branch **Bb** itself. Then predicting this block with the two-block ahead branch target buffer presented above is likely to result in many misfetches. Yeh et al. reached the same conclusion and proposed the fetching of only a single block in this case [19].

A specific solution for coping with predicting the instruction block following the return target block is presented here.

Second Address Stack As already mentioned, the address **Di** of the instruction block following the return target is more dependent on the address of the return target **Ci** than on the return address **Bb** itself. A specific difficulty is that the return target is unknown when its block successor has to be predicted.

Nevertheless, a branch strongly associated with the return target **Ci** has been already issued in the instruction flow: the procedure call **Pp= Ci**-1. We propose associating the information on the instruction block **Di** with a BTB entry associated with the procedure call **Pp**.

This is illustrated in figure 5. A special entry type **R** is introduced in the Two-Block Ahead BTB for dealing with this case.

A BTB entry PpR is allocated instead of an entry BbT when a branch in block C is mispredicted to keep information about the branch Cc (figure 5 (c)). It should be noted that information Pp can be easily recovered because Pp equals Ci-1.

- A BTB entry **PpR** is allocated instead of an entry **BbT** when a branch in block **C** is mispredicted to keep information about the branch **Cc** (figure 5 (c)). One should note that information **Pp** can be easily derived since **Pp** equals **Ci**-1.
- The BTB is searched for an entry **PpR** whenever a call instruction **Pp** is fetched (figure 5 (a)). This information must be kept until the return instruction **Bb** is fetched. For this purpose, the two-block ahead branch predictor uses a Second Address Stack (SAS). A copy of the **PpR** entry is pushed into the SAS whenever **PpR** hits in the BTB. Otherwise, an invalid entry is pushed. Notice that the same number of entries are pushed in the return Address Stack and in the SAS.
- When a return is popped from the Return Address Stack, an entry is popped from the SAS to accurately predict a branch in the subsequent basic block if any (figure 5 (b)). Notice that this branch may be a return.

The whole process is further detailed in figure 4.

4.4 A further optimization

When using the presented two-block ahead branch predictor, and when a branch **Bb** is predicted not taken, the following predicted instruction block begins at address **Bb+1**.

But **Bb** may not be the last instruction in the block which was read in parallel from the I-cache. For instance, in the example illustrated in figure 6 (a), three consecutive fetches are issued on the same cache block. Instruction blocks E, F and G are read in parallel on the first fetch, but blocks F and G are then discarded.

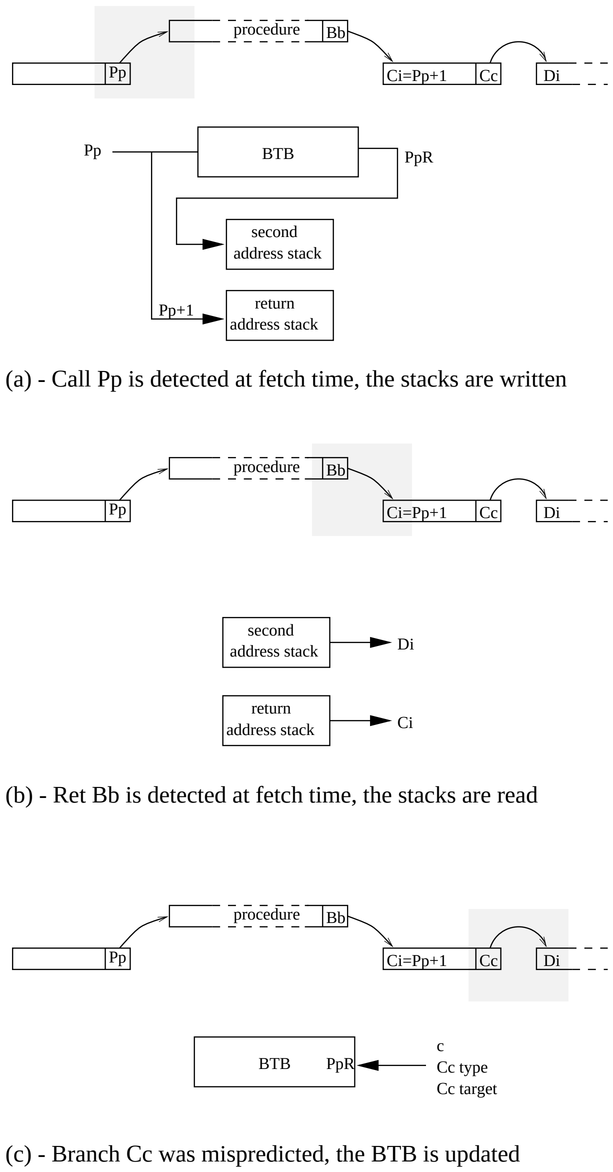


Figure 5: The two-block ahead return stack.

Such a situation wastes I-cache bandwidth. Being able to pick at the same time all the useful consecutive instructions read in parallel (i.e. instruction blocks E,F and G in the illustrated example) in an instruction cache block would obviously save many fetch cycles.

A complex general case Several consecutive conditional branches may lie in the block **Bi** of instructions read in parallel. Ideally, the instruction fetch mechanism should be able to forward the entire sequence of consecutive useful instructions in this block for further processing , then bypass all the consecutive not-taken branches (figure 6 (b)).

However, predicting the instruction block fetched after this sequence is a rather challenging problem when using information associated only with the predecessor instruction block **Aa**:

- The fetched block may be any of the targets of the consecutive conditional branches or it may also be the fall-through block.
- A branch prediction must be performed for each one of the conditional branches in the cache block.

We have not yet been able to find a simple solution for this general case although collapsing may lead to a solution.

Bypassing the last branch in the instruction block When **Bb** is the last branch in the fetched instruction block and is not-taken, then the successor block of **Bb** is the block beginning at address **(B+1)**0..0.

Let us suppose that the information **L** "the branch is the last one in the cache block (or not)" is recorded in the BTB entry. Then the instruction-address generator can use this information to compute the predicted instruction block as follows. When **L** is set and the branch **Bb** is predicted not-taken, the predicted instruction block is the block beginning at at address **(B+1)**0..0 instead of the block beginning at address **Bb+1** which is fetched during this cycle.

The extra hardware required for implementing this optimization is quite low: an extra bit in each BTB entry and some logic in the decode stage for computing **L**. On the other hand, it systematically saves one instruction block fetch when it applies (figure 6 (c)). It should be noted that this solution may also be adapted to conventional branch target buffers, and that its efficiency may be highly improved with software ordering of most likely taken branches to be fall-through.

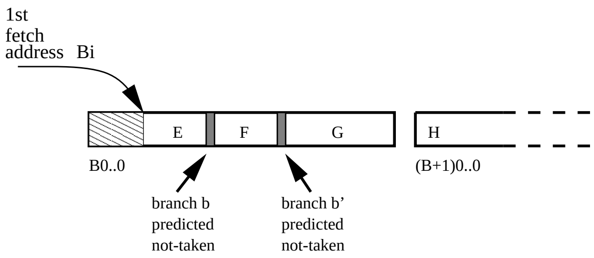
4.5 Two-Block Ahead Branch Predictor in a double I-fetch processor

When using a double I-fetch processor, the I-cache must either be fully double-ported or interleaved [18].

When using the two-block ahead branch predictor, the branch prediction table and the branch target buffer must also be either fully double-ported or interleaved.

When the I-cache is interleaved, the branch prediction table and the branch target buffer may be also interleaved in the same way. That is when blocks **A** and **B** are conflicting on the I-cache, they are also conflicting on the branch prediction table and the branch target buffer (and vice-versa). In this case, using an interleaved two-block ahead branch predictor will not impair performance at all.

Furthermore, the RAS and SAS must be able to deliver two addresses per cycle:



(a) worst case

	fetch address	fetched instruction groups
1st fetch	Bi	E
2nd fetch	Bb+1	F
3rd fetch	Bb'+1	G
4th fetch	(B+1)0..0	H

(b) ideal case

	fetch address	fetched instruction groups
1st fetch	Bi	EFG
2nd fetch	(B+1)0..0	H

(c) branch b' bypassed

	fetch address	fetched instruction groups
1st fetch	Bi	E
2nd fetch	Bb+1	FG
3rd fetch	(B+1)0..0	H

Figure 6: Bypassing not-taken branches.

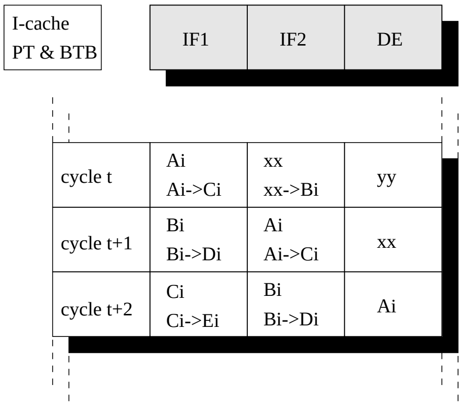


Figure 7: Pipelined Address Generation Timings.

- When **Bb→Ci** is a return, the return stack must deliver **Ci** and the transition **Cc→Di**. In this case, the SAS is used to compute **Di**.

  When transition **C→Di** is also a return, a second read is done on the return stack.
- When **Aa→Bi** is a return, the SAS is used to compute **Ci**. When **Bb→Ci** is also a return, a second read of the SAS is used to compute **Di**.

When both transitions **Bb→Ci** and **Cc→Di** are calls, these two stacks have also to be able to accept two pushes per cycles.

5 Single I-Fetch Processors and Two-Block Ahead Branch Prediction

In Section 6, we will show that branch prediction information can be associated with the previous branch instructions without degrading the prediction accuracy. With such a scheme, two addresses are predicted in a single cycle in double I-fetch "brainiac" processors as shown in the previous section. Instead of exploiting more parallelism, another way to get performance improvement is to increase the clock rate, leading to single I-fetch "speed-demon" processors. In this section, we first show that the instruction-address generator stage may be the critical path of the processor, then we show that pipelining the instruction-address generation process (figure 7) in such single I-fetch processor can be done by means of the two-block ahead branch predictor.

5.1 Instruction Address Generation may be a Critical Path

In current single I-fetch processors, both the I-cache and the branch predictor are accessed with the current instruction block starting address. By the end of the cycle, the starting address of the next instruction block must be generated. In some of the processors, the I-cache access time is longer than the cycle time. For instance, the Intel PentiumPro features a pipelined I-cache access completed within two cycles.

As far as the current instruction block address is used to predict the next instruction block, either the instruction address generator can compute the starting address of the next instruction block in a single cycle, or bubbles are inserted in the pipeline in the case of branches as in the Intel PentiumPro. Indeed, accessing the prediction structures in the PentiumPro is spread over two cycles, mainly because its structures feature a high number of entries. Reducing the number of entries impairs the performance, especially on such an heavy-pipelined processor. The instruction-address generation process is quite complex because it includes several consecutive steps:

1. Parallel accesses to the BTB, the Prediction Table (PT), and the return address stack. Computation of the fall-through address.
2. Prediction of the outcome and selection of the generated address. Possible updates of the return stack and the branch history register.

In particular, the read of a set-associative BTB featuring a high number of entries is time consuming. Achieving the complete instruction address generation process in a single cycle may be a challenging problem in high clock-speed processors. The instruction address generator might then be the critical electrical path, determining the processor cycle time.

5.2 Pipelining the Instruction Address Generation Process

A two-stage pipelined branch predictor is depicted in figure 8. The BTB and PT illustrated in this figure are two-block ahead BTB and PT implementations computing only one address in a single cycle.

Let **Ai**, **Bi**, and **Ci** be the starting addresses of the instruction blocks respectively fetched at cycle t, t+1, and t+2, **Aa**, **Bb**, and **Cc** be the addresses of the last instruction in these blocks, **A**, **B**, and **C** be these block numbers, **Ha**, **Hb**, and **Hc** be the branch history registers during cycle t, t+1, and t+2. The behavior of the pipelined address generator is represented in figure 7 and is as follows:

1. **cycle t**: In first stage IF1 of the pipeline, **A** and **Ha** are used to index the PT and the BTB. The fetching of the instruction block **A** begins in the instruction cache. At the end of the cycle, **Bi** and **a** flow out from the second stage IF2 as does the type of the transition from **Aa** to **Bi** (**T**, **N**, or **R**).
2. **cycle t+1**: In stage IF1, **B** and **Hb** are used to index the PT and the BTB while the fetching of **B** begins in the instruction cache. In stage IF2, the access to the instruction cache, and to the BTB and the PT with **A** and **Ha** are completed. In particular, the tag check on the BTB is performed during this cycle.

   The fall-through address for block **B** is also computed (**Bb**+1). Depending on the type of the transition from **Aa** to **Bi**, and on the information flowing out from the BTB and the PT, **Ci** is chosen among four addresses (the target addresses flowing out from the BTB, top of the RAS or top of the SAS, or the fall-through address **Bb**+1) as related in the first algorithm of figure 4. By the end of cycle t+1, position **b** and transition type **Bb→Ci** are forwarded to proceed the tag-matching process on the next cycle.
3. **cycle t+2**:**C** is used to index the instruction cache and to compute the two-block ahead instruction block starting address in stage IF1. The instruction block **C** is decoded.

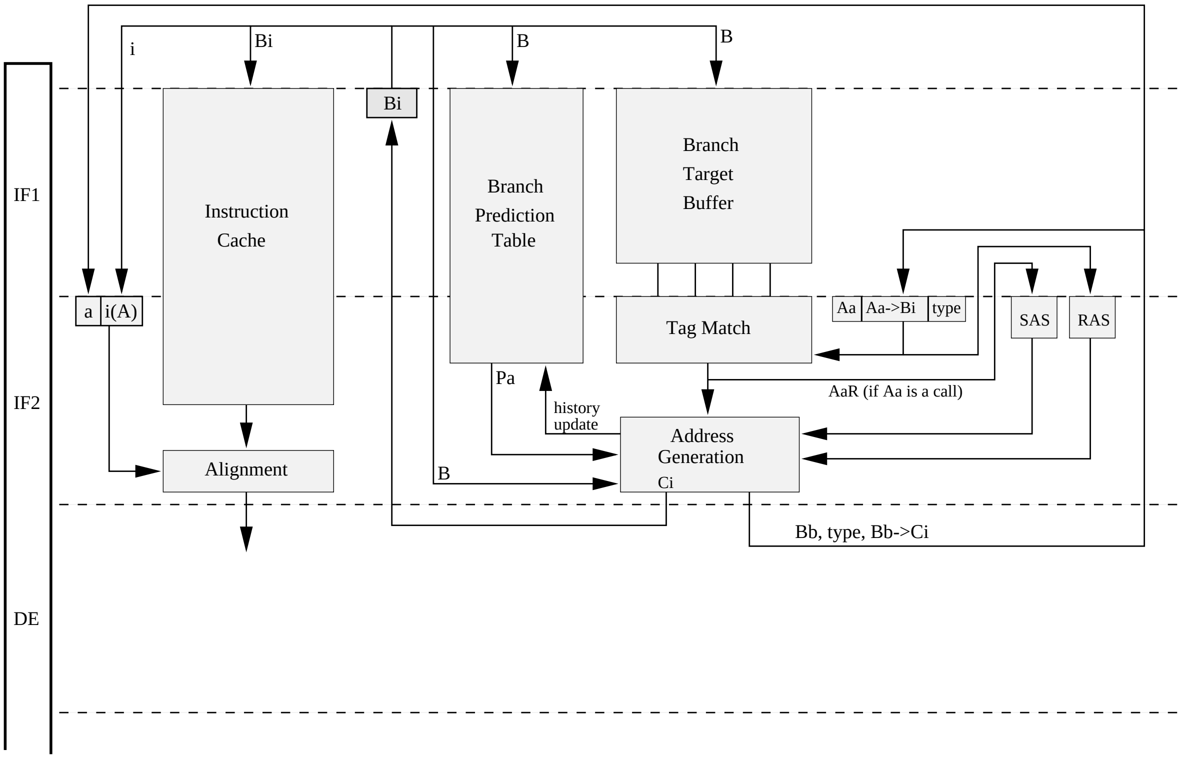


Figure 8: The Two-Stage Pipelined Branch Predictor (cycle t+1).

One should note that some information (the type of the transition from **Aa** to **Bi**, the position **a** of the last instruction in block **Ai**) produced during cycle t are used during cycle t+1. Nevertheless, these items are not critical. Basically, during the instruction-address generation process, the most time consuming actions are the accesses to the BTB and the PT. Therefore, pipelining the instruction address generation process as described above allows the use of a shorter cycle than with conventional address generators and/or to implement bigger prediction tables without introducing any one-cycle penalty in the presence of a predicted taken branch.

Misprediction penalty Using a two-block ahead branch predictor in an out-of-order single I-fetch processor like the Intel PentiumPro or the MIPS R10000 does not result in a one-cycle increase of the misprediction penalty. As a matter of fact, the address of the non-predicted path is recorded in the checkpoint established for the branch to resume fetching in processors featuring a one-block ahead branch predictor. The two-block ahead scheme only requires to record in addition the non-predicted path in the IF2 stage (**AaT** or **AaN** if branch **Aa** was mispredicted) and the prediction made **Pa**, since all the other information required in stage IF2 to compute **Ci** would be known (fall-through address and return stack values).

In in-order single I-fetch processors, a structure can hold such values. Otherwise, the misprediction penalty would be increased by one cycle. This extra cycle is required to retrieve both the prediction and the possible addresses of the target instruction **Ci**. Nevertheless, one should note that replacing an instruction address generator resulting in pipeline bubbles on branches as in the DEC 21164 by our predictor would save the bubble in all cycles where the prediction is correct. For incorrect predictions, the penalty is the same for both mechanisms.

6 Experimental Results

Trace-driven simulations were conducted to verify the effectiveness of the two-block ahead branch predictor. We first establish that the branch prediction accuracy achieved by our branch prediction table is equivalent to those obtained with conventional one-block ahead branch prediction tables. Finally, we show that the two-block ahead branch predictor does not require any additional BTB logic to handle most branches.

6.1 Branch Prediction Accuracy

We assumed a perfect BTB (all branches hit) in these simulations to compare predictors without any clouding effects from the BTB. The simulations were run only over the CINT92 suite since floating-point benchmarks tend to lower misprediction rates.

Figure 9 (a) presents the average misprediction rate for two common prediction schemes with respect to the size of the prediction table. The misprediction rates are reported for both the two-block ahead branch predictor (*g-share 2*, *g-select 2*) and the corresponding one-block ahead branch predictor (*g-share*, *g-select*). These branch prediction schemes differ by the index which is used to access the prediction table. The prediction table in *g-select* is indexed with a concatenation of branch history and branch address bits. The index value in *g-*

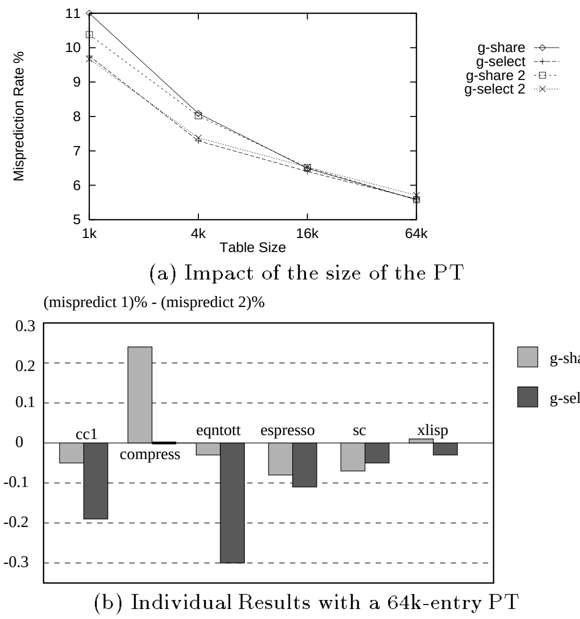


Figure 9: Branch Misprediction Rates.

share is the exclusive OR of the branch address with the branch history register.

Notice that, for both schemes and for all table sizes, the performance of the two-block ahead branch predictors is very close to the performance of the corresponding one-block ahead branch predictors. The difference between the misprediction rates of the different benchmarks are reported in figure 9 (b) for a 64 K-entry prediction table. These differences are very tiny and do not exceed 0.30 %.

From these simulation results, we conclude that the two-block ahead branch history register and the two-block ahead address are as representative of a branch as the conventional branch history register and the branch address.

6.2 The Branch Target Buffer

In the previous sections, we have introduced the two-block ahead branch target buffer to predict two block addresses per cycle or to pipeline the address generation process. Here our results verify that such a mechanism does not require a high degree of associativity or a large number of entries since a branch can be associated with more than one BTB entry.

Figure 10 reports the individual results with a 512-entry and a 2K-entry BTB with varying degrees of associativity. A pseudo-random replacement policy was used and the cache line size was assumed to be 16 instructions wide as in the MIPS R10000 and most of the current out-of-order processors. All the branches in the same cache line map to the same set in the BTB. In addition in the two-block ahead BTB, different types of branches may be associated with the same address tag (**Aa**T and **Aa**N for instance). Thus a set-associative BTB is required.

We can see from figures 10 (a) and 10 (b) that the maximum hit rate is nearly reached with an associativity of 4, compared to an associativity of 2 for a conventional BTB (figures 10 (c) and 10 (d)). These results also show that for realistic BTB sizes, the hit rate for a conventional BTB is slightly better than that for a two-block ahead BTB. However, this difference is less than 0.5 % for most applications (including gcc), so the improvement of fetching two blocks per cycle is still valuable. Thus, the two-block ahead branch predictor does not require any additional storage in the BTB, nor does it lead to any increase of the associativity.

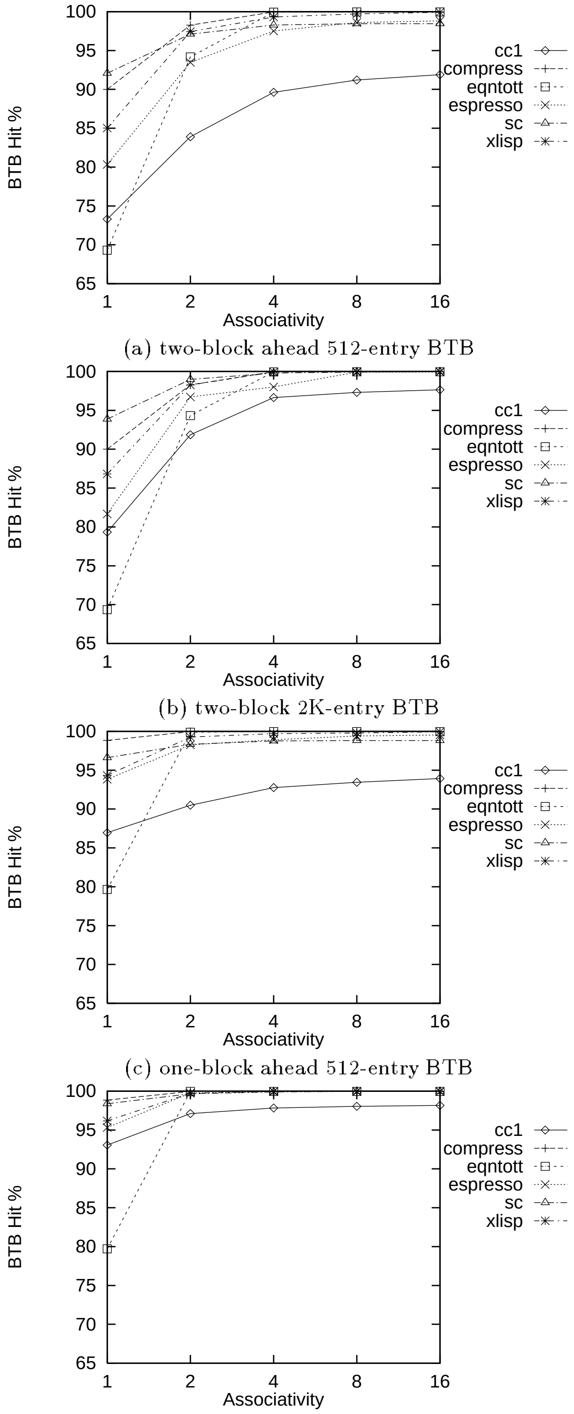
7 Summary and Concluding Remarks

The current instruction-fetch mechanisms limit the performance that may be achieved. New solutions must be implemented in next generation microprocessors.

Two design philosophies have been used to achieve higher performance for the past four years. “Brainiac” processors attempt to achieve the highest level of IPC possible. Future generation “brainiac” processors should fetch more than one basic block in a single cycle, otherwise the fetch limit of one basic block per cycle would significantly impair performance. This raises the difficult issue of predicting multiple instruction blocks in parallel. On the other hand, “speed demon” processors get high-performance by increasing the clock rate. All parts of the processor must be pipelined and some functions are spread over several cycles (e.g. I-cache access). However the address-generation process is not pipelined in current designs. In these processors, either the address-generation mechanism (including branch prediction) becomes the electrical critical path or pipeline bubbles are inserted for each predicted taken branch. As this may severly limit the performance achieved in future designs, pipelining the address generation and the branch prediction is also a major issue.

We have introduced the Two-Block Ahead Branch Predictor to deal with both issues. In conventional branch prediction mechanisms, information associated with the current instruction block such as the address of the branch instruction is used to predict the next instruction block. The two-block ahead branch predictor uses the same information when predicting the block following the next instruction block. The amount of information stored in our predictor is in the same range as in a conventional branch prediction mechanism. Furthermore, any branch prediction schemes proposed for single I-fetch processors can be adapted to our predictor. Simulations have shown that equivalent branch prediction accuracy is achieved. Thus, the two-block ahead branch predictor can be used to predict the address of two basic blocks in a single cycle, improving the hardware ILP of “brainiac” processors. It can also be used to pipeline the address-generation process over two cycles in “speed-demon” processors. The prediction accuracy remains the same.

The two-block ahead branch predictor can be extended to a multiple-block ahead branch predictor fetching multiple basic blocks in a single cycle or to further pipeline the address generation process over more than two cycles. We plan to study how the scheme scales from two-block to multiple-block ahead, especially on the return address structures, and on the accuracy and the features of the prediction structures. Any combination between the multiple-block ahead branch predictor and the previously proposed multiple schemes [19, 4] may worth be investigated to implement a high-end fetch mechanism or to pipeline a double I-fetch processor for instance.



(a) two-block ahead 512-entry BTB

(b) two-block 2K-entry BTB

(c) one-block ahead 512-entry BTB

(d) one-block 2K-entry BTB

Figure 10: BTB Hit Rates According to the Associativity and the Number of Entries.

The structures of the two-block ahead BTB and the two-block ahead PT presented in this paper have been directly deduced from existing conventional one-block ahead solutions. We are now investigating specific implementations of those two-block ahead structures. For instance, the double I-fetch implementation of our predictor features two dual-ported memory structures. We are looking at ways to build fast and cost effective structures by taking into account the correlation between basic blocks. We are also looking at ways to adapt cost-effective solutions for one-block ahead BTBs [2, 15] to two-block ahead BTBs.

Acknowledgments

We gratefully acknowledge the help and encouragement of the members of the HPS research group at ACAL, University of Michigan, during our work on this project, in particular, Eric Hao, Sanjay Patel, Daniel Friendly, Paul Racunas, Lee Hwang Lee, Tse Hao Hsing, Darren Vengroff, and Professor Yale Patt. They gave many inputs and provided critical comments on early versions of the paper. We also thank Richard Uhlig, presently at IRISA, for his help in polishing the final version, and Andy Glew (Intel) and the anonymous reviewers for their insightful comments.

References

[1] **M. Butler and Y. N. Patt,** "An Investigation of the Performance of Various Dynamic Scheduling Techniques," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992.

[2] **B. Calder and D. Grunwald,** "Next Cache Line and Set Prediction," *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[3] **T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel,** "Optimization of Instruction Fetch Mechanisms for High Issue Rates," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[4] **S. Dutta and M. Franklin,** "Control Flow Prediction with Tree-Like Subgraphs for Superscalar Processors," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[5] **L. Gwennap,** "Digital Leads the Pack with 21164," *Microprocessor Report*, September 1994.

[6] **L. Gwennap,** "Comparing RISC Microprocessors," *Proceedings of the Microprocessor Forum*, October 1994.

[7] **L. Gwennap,** "PA-8000 Combines Complexity and Speed," *Microprocessor Report*, November 1994.

[8] **L. Gwennap,** "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report*, February 1995.

[9] **IBM and Motorola,** "PowerPC 604 RISC Microprocessor User's Manual," MPR604UMU-01, 1994.

[10] **M. Johnson,** Superscalar Microprocessor Design, *Prentice-Hall*, 1991.

[11] **S. Jourdan, P. Sainrat, and D. Litaize,** "An Investigation of the Performance of Various Instruction-Issue Buffer Topologies" *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995.

[12] **S. McFarling,** "Combining Branch Predictors," *Technical Note TN-36*, DEC-WRL, June 1993.

[13] **Mips Technologies Incorporated,** "R10000 Microprocessor Product Overview," *Technical Report*, October 1994.

[14] **D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi,** "Control Flow Prediction for Dynamic ILP Processors," *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993.

[15] **A. Seznec,** "Don't use the page number, but a pointer to it," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[16] **M. D. Smith,** "Tracing with Pixie," *Technical report*, Stanford University, April 1991.

[17] **SPEC 92,** *Technical report*, December 1992.

[18] **S. Weiss and J. E. Smith,** POWER and PowerPC: Principles, Architecture and Implementation, *Morgan Kaufmann Publishers Inc.*, 1994.

[19] **T. Yeh, D. T. Marr, and Y. N. Patt,** "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache," *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[20] **T. Yeh,** "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," *PhD thesis*, Department of Electrical Engineering and Computer Science, University of Michigan, 1993.