Effective ahead pipelining of instruction block address generation*

André Seznec and Antony Fraboulet
IRISA/INRIA
Rennes, France
{seznec,afraboul}@irisa.fr

Abstract

On a N-way issue superscalar processor, the front end instruction fetch engine must deliver instructions to the execution core at a sustained rate higher than N instructions per cycle. This means that the instruction address generator/predictor (IAG) has to predict the instruction flow at an even higher rate while the prediction accuracy can not be sacrificed.

Achieving high accuracy on this prediction becomes more and more critical since the overall pipeline is becoming deeper and deeper with each new generation of processors. Then very complex IAGs featuring different predictors for jumps, returns, conditional and unconditional branches and complex logic are used. Usually, the IAG uses information (branch histories, fetch addresses, ...) available at a cycle to predict the next fetch address(es). Unfortunately, a complex IAG cannot deliver a prediction within a short cycle. Therefore, processors rely on a hierarchy of IAGs with increasing accuracies but also increasing latencies: the accurate but slow IAG is used to correct the fast, but less accurate IAG. A significant part of the potential instruction bandwidth is often wasted in pipeline bubbles due to these corrections.

As an alternative to the use of a hierarchy of IAGs, it is possible to initiate the instruction address generation several cycles ahead of its use. In this paper, we explore in details such an ahead pipelined IAG. The example illustrated in this paper shows that, even when the instruction address generation is (partially) initiated five cycles ahead of its use, it is possible to reach approximately the same prediction accuracy as the one of a conventional one-block ahead complex IAG. The solution presented in this paper allows to deliver a sustained address generation rate close to one instruction block per cycle with state-of-the art accuracy.

1 Introduction

Nowadays a state-of-the-art instruction address generator (IAG) on a superscalar processor features a conditional branch predictor, a jump predictor, a return stack, a branch target buffer (BTB) or branch target computation hardware, sequential block address computation and a final address selection stage (for instance the IAG on Alpha EV8 [12]). So far, IAGs use information (branch histories, fetch addresses, ...) available at a cycle to predict the next fetch address. The response time of a complex IAG is longer than a single cycle since the mid 90's (e.g. PentiumPro [5], Alpha 21264 [7] and EV8 [12]). The solution that has been used so far is to rely on a hierarchy of IAGs. A fast IAG performs a prediction in a single cycle, while a second slower but more accurate IAG performs a (hopefully) more accurate instruction block prediction in two or more cycles. Whenever the two predictions disagree, bubbles are inserted in the front-end pipeline, and the instruction fetch resumes with the result of the slow IAG. Such a strategy leads to the waste of a significant part of the instruction fetch bandwidth. These solutions based on a hierarchy of IAGs are not scaling with the technology. The size of the tables that can be read in a single cycle becomes smaller and smaller [1] while complex indexing schemes and extra logic are required (e.g. [12]). Therefore the accuracy of fast IAG prediction will reduce with each new processor generation, while the response time of the state-of-the-art IAG will become comparatively longer and longer. This will generate more and more bubbles in the front-end pipeline.

As an alternative solution to this hierarchy of predictors, Seznec et al. [13] proposed to use two-block ahead branch prediction. Information associated with an instruction block is used to predict the two-block ahead instruction block address instead of the standard one-block ahead instruction block address. This allows to pipeline the instruction address generator on two cycles. Recently Jimenez [6] showed that ahead pipelining a simple *gshare* branch predictor may result in a higher fetch bandwidth than using one-cycle ahead complex conditional branch predictors with multiple cycle response time.

In this paper, we investigate in detail a general and aggressive form of ahead pipelining of the complete IAG. All the components of the IAG do not have to be pipelined

*This work was partially supported by an Intel research grant

on the same depth (e.g. the conditional branch predictor read is initiated five cycles ahead while the jump predictor is read only three cycles ahead). When the instruction address generation is initiated a few cycles ahead, a small amount of inflight information on the intermediate instruction fetch blocks can be introduced in the last stages of the IAG. Predicting the decode information needed by the IAG (presence/absence and types of control flow instructions) is needed to avoid misfetches, i.e. address block predictions that are corrected at decode time. We present a very efficient combination of mechanisms to accurately predict this decode information. We also show that a medium-sized volume of back-up information in the checkpoint/repair mechanism (in the same range as the instruction volume itself) allows a smooth restart of the instruction address generation after a misprediction despite the five-cycle pipelined IAG.

The remainder of the paper is organized as follows. In Section 2, we present the spectrum of instruction block definitions we are exploring and the technological hypothesis done in this study. Section 3 presents a base design for a state-of-the-art hierarchy of IAGs (derived from the EV8 design [12]). Section 4 details the different issues associated with ahead pipelining the instruction address generator. Section 5 presents the performance evaluation framework and metrics. Section 6 compares the performance of a hierarchy of IAGs and an ahead pipelined IAG. We point out that the hierarchy of IAGs wastes a very significant part of the instruction generation bandwidth in the pipeline bubbles inserted when the slow and fast IAGs disagree. The ahead pipelined IAG allows an overall accuracy of instruction block address prediction close to the one of the hierarchy of IAGs. In average, it encounters only about 1/10th to 1/5th of the misfetches suffered by the hierarchy of IAGs. Therefore it allows to reach an effective IAG throughput close to one instruction block per cycle. Finally, Section 7 summarizes this study and presents directions for future works.

2 General framework

Several studies [16, 13, 15, 2] have considered predicting the addresses of two or more non-contiguous instruction blocks in parallel. The abandoned Alpha EV8 [3] is fetching (up to) two non-contiguous instruction blocks per cycle. The study presented in this paper is limited to the fetching a block of contiguous instructions per cycle.

Reinman et al [10] proposed to decouple the Instruction Address Generation from the instruction fetch through an instruction fetch queue. This allows to generate instruction address for blocks wider than the maximum instruction fetch bandwidth. It also allows to hide part of the latency associated with the use of a hierarchy of IAGs. The Instruction Address Generator we are considering this paper is strongly coupled with the rest of the instruction front-end: on each cycle, the address of the block to be fetched on the next cycle is generated. Decoupling instruction address generation from instruction fetch will be considered in future studies.

In this section, we first present the spectrum of the definitions of an instruction fetch block we are exploring in this paper. Then we present the technological assumptions that were made in this paper.

2.1 Various instruction fetch block definitions

There are various possibilities for the definition of an instruction fetch block depending on the maximum number of instructions fetched in parallel, on the physical frontier of an instruction block and how not-taken conditional branches are handled. These various definitions are illustrated on Figure 1 with all conditional branches assumed to be not-taken.

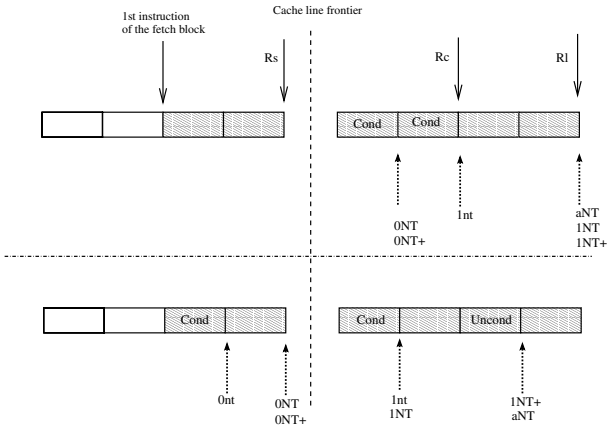

Figure 1. Various definitions of instruction fetch blocks

Maximum range of an instruction block In this paper, we consider three possibilities for the maximum range of an instruction cache block. CL being the cache block size:

- **Rs** (for short range): a fetch block must end on the frontier of the cache block.
- **Rc** (for cache block size range): a fetch block features a maximum of CL instructions.
- **Rl** (for long range): a fetch block must end on the frontier of the second cache line.

Rc and **Rl** implicitly assume that the instruction cache is bank-interleaved. **Rc** and **Rs** allow to fetch up to CL instructions per cycle while **Rl** allows fetch up to $2 * CL$ instructions per cycle.

Ending an instruction block Apart from the maximum range ending condition, an instruction fetch block must end on a taken control-flow instruction, but not necessarily on a not-taken conditional branch. We consider the three following basic policies illustrated on Figure 1 for **Rl**.

- **0nt**: an instruction fetch block always ends on the first control-flow instruction.

- **1nt**: the first not-taken branch is bypassed, but the second control-flow instruction always ends the fetch block.
- **aNT**: all not-taken branches are bypassed.

A simple optimization can be applied on policies **0nt** and **1nt**. If the "normal" last instruction in the block is a not-taken branch and if completing the fetch block by applying the maximum range condition (for instance **Rl**) does not add any control flow in the fetch block, then the instruction fetch block is completed by applying this maximum range condition. These simple optimizations will be referred to as **0NT** and **1NT**.

For the IAGs considered in the paper, return and jump targets are predicted using independent jump predictor and return address stack. If we assume a fetch block ending with a not-taken branch **0NT** or **1NT** and if the next control flow instruction is either a return or a jump then the prediction of the address of the block after this jump or return is straightforward (provided one is able to recognize the presence of the return or jump). The same way, if the next control flow instruction is an unconditional branch then there is no need to predict the direction of the condition. If one is able to predict or compute an extra target then one is also able to predict this unconditional branch. We will refer to the policies where the fetch block is extended up to the next unconditional control flow instruction as **0NT+** and **1NT+** respectively.

0NT or **0nt** has been implemented in most processors so far. **1nt** was considered in [8]. It was shown that it allows to achieve about 25 % increase in the overall instruction fetch bandwidth over **0nt** assuming **Rl**. **aNT** (associated with **Rs**) is used on the Alpha EV8 with 8-instruction cache lines.

2.2 Technological hypothesis

The technological hypothesis we are using in this paper are derived from those found on the Alpha EV8 instruction front-end design [12], but assume a more aggressive clock. For the purpose of this study, we suppose that the target frequency is twice the frequency of the Alpha EV8 processor for the same technology. The technological hypothesis considered in the paper are extrapolated from the EV8 design and are assuming a twice shorter cycle, i.e., our cycle corresponds to a single phase on EV8. We will consider that 1) Read of a 16Kb non-tagged table and fast-forwarding the read result for use as the next index consumes two cycles (wordline selection, followed by column selection) and we will assume that if one wants to guarantee a single cycle read access time on a table then the size of the table must be equal or smaller than 2 Kb. 2) The final selection of an address through a 8 to 10 entries multiplexor consumes a cycle. *This corresponds approximately to the Alpha EV8 final selection in the instruction address generator [12].*

3 Base design

The base conventional IAG we are considering in this paper is illustrated on Figure 2. This IAG is derived from the hierarchical IAG of the Alpha EV8 [12], apart that we only consider fetching one instruction block per cycle.

The Alpha EV8 processor features a single cycle line predictor and a two-cycle complex instruction address generator (IAG). IAG consists in a return stack, a jump predictor, a conditional branch predictor, a branch target computation engine (a possible target is directly derived from the instruction flowing out from the instruction cache for each instruction), fall-through computation and a final selector. The final selector selects the address of next instruction block among the eleven possible instruction block addresses during the second phase of the second cycle. The control of the final selector is obtained by a combination of the branch conditional prediction and some decode information from the instructions in the fetch block (presence/absence of a control-flow instruction, type of the control-flow instruction).

Under the technology assumptions we have listed in the previous section, in this paper, we consider an IAG similar to the one found on Alpha EV8 [12], but pipelined on 4 cycles (the conditional branch predictor table reads span over 3 cycles). When slow IAG begins computing address of block B, information available to index the conditional predictor table consists of address A (output of the line predictor) and history from three-block ahead history W, inflight information from intermediate blocks X, Y and Z are injected inflight in the indices of the tables (see Section 4.1.1).

Under our technological hypothesis, single cycle line prediction can only be implemented with a very small table (in the 2Kb range). Similar to the prediction of the first of the two fetch blocks on Alpha EV8, we assume that the line predictor works as follows: a 1-bit direction prediction is read concurrently with the line predictor and used as a selector between the address read on the line predictor and the fall-through block. This direction prediction table features four times more entries than the address table as on Alpha EV8. The accuracy of the line predictor accuracy is relatively poor. As an alternative, we also considered a 16Kb two-block ahead line predictor derived from two-block ahead branch prediction [13], but experiments showed that it performed worse than the small line predictor in average.

In case of a mismatch between line prediction and instruction address generation, three bubble cycles are inserted in the front-end instruction fetch.

4 Ahead pipelining the instruction address generator

The goal of ahead pipelining the IAG is to avoid the pipeline bubbles inserted in the front-end by line mispredic-

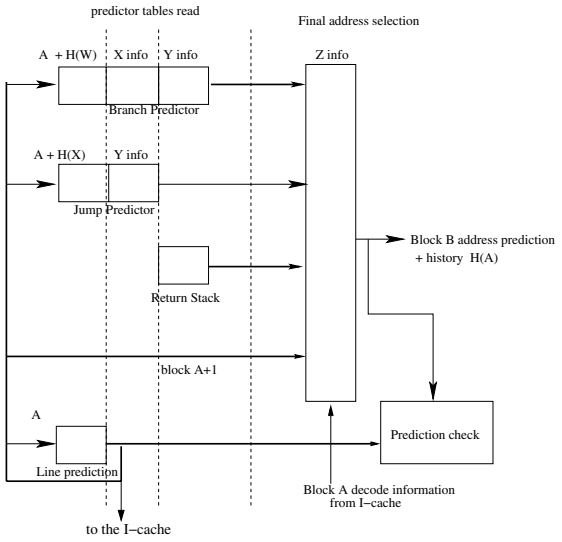

Figure 2. Hierarchical instruction address generator

tions while retaining the same accuracy as a state-of-the-art IAG.

The ahead pipelined IAG we are proposing in this section is illustrated in Figure 3. The respective timings of the ahead pipelined IAG and of the hierarchy of IAGs are illustrated on Figure 4: the instruction address generation for block is initiated five cycles ahead on the ahead pipelined AHIAG instead of one cycle ahead on the hierarchy of IAGs.

The ahead pipelined IAG features mechanisms with similar functionalities as the ones found as the hierarchical IAG: return address stack, jump predictor, sequential block address computation, conditional branch selection and final selection stage. In order to replace the branch target computation hardware that uses information flowing from the instruction cache, a BTB is used. On the hierarchical IAG, all the tables are read using information on the last instruction block as indices. On the ahead pipelined IAG, the access to the tables in these components is initiated in advance at different cycles depending on their read access time. The information used to initiate the read of the tables are the most recent available instruction block addresses and branch information, but since the read spans over several cycles, additional information can be used in the indices (for wordline selection for instance).

As on the hierarchical IAG, decode information (presence/absence and type of control flow instructions in the current instruction block) is needed for controlling the final address selection on the ahead pipelined IAG. On the hierarchy of IAG, the real decode information is available for the slow IAG (from the I-cache). This real information is not available in time on for the ahead pipelined IAG and therefore must be predicted. Mispredictions due to incorrect speculative decode information are detected at decode time in the pipeline and then can be corrected at this time. The penalty induced on such a misprediction is equivalent to the penalty on a line misprediction for the base design. As a consequence, ahead pipelining the IAG is an effective solution to increase the instruction fetch only if the prediction of the decode information is significantly more accurate than line prediction.

On a misprediction, the instruction address generation must resume with the correct instruction block address, decode information and branch history. The ahead pipelined IAG cannot compute the address of the block following the mispredicted block in less than than five cycles, thus an extra 4-cycle mispenalty can be encountered. We will show in Section 4.3 that, by storing a medium amount of extra information in the checkpoint repair mechanism, a full speed restart of instruction address generation is possible.

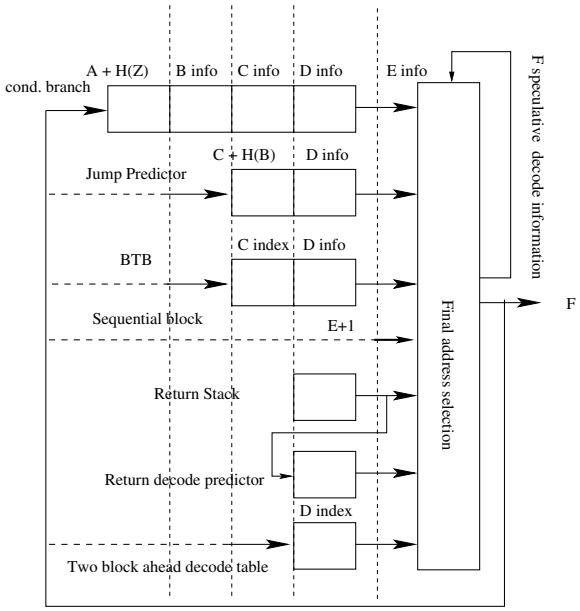

Figure 3. Ahead pipelined instruction address generator

4.1 Predicting targets and branch directions

4.1.1 Inflight intermediate information incorporation

On the EV8 branch predictor [12], the read of branch prediction tables span over three clock phases. The read consists of three consecutive steps, involving three different bit subsets in the index, first wordline selection, second column selection, and finally unshuffle permutation[1]. The constraints on the three different parts of the index were different: immediate availability for the wordline bits, one phase to compute the column bits and a full cycle to compute the unshuffle permutation bits.

[1] a 8-to-8 permutation is performed

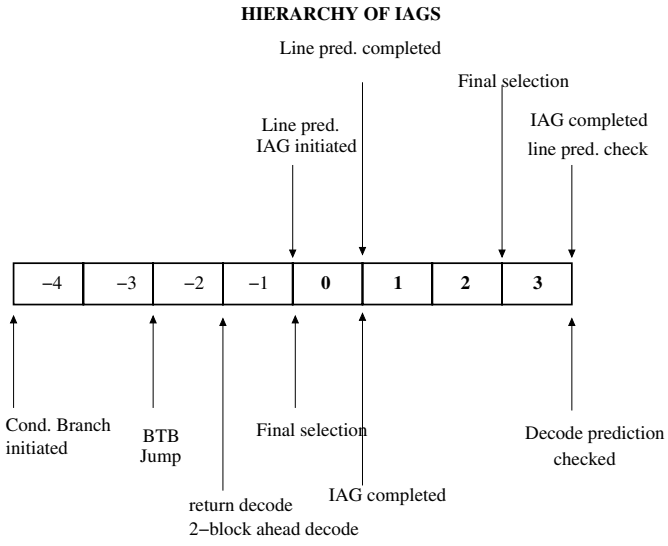

Figure 4. Respective timing on the ahead pipelined IAG and the hierarchical IAG

On the ahead pipelined IAG, we also leverage this flexibility: information that is unknown at the initiation of the read of a table can be used inflight for computing part of the index for a table read (for example the wordline index).

Even if the access to the conditional branch predictor for ahead instruction block address generation of block F begins five-cycle ahead i.e, when only address of the block A is known and history from block Z, the indices used to read the tables may incorporate information from all the blocks B, C, D and E.

4.1.2 Final selection

The final selection of the next block address among the exits of the components of the instruction address generator is assumed to consume one cycle. We assume that, all addresses flowing from the return stack, jump predictor and branch target buffers are available at the beginning of the cycle. We also assume that, all the information bits needed for the control of the final selector are available at the same time. But we assume that the computation of the address of the sequential block can be computed in parallel with the final selection.

Incorporation of one-bit inflight information on the last predicted fetch block is done as follows: two sets of addresses are generated by the jump predictor and branch target buffer and two sets of branch predictions are generated, the one-bit inflight information is used to select among these sets.

4.1.3 Predicting conditional branches

Conditional branch prediction result is needed to control the final address selection. Therefore the conditional branch prediction results for the branches in block D must be available at the same time of all the possible addresses at the entry of the final address selection stage, i.e. at the same time as address of block D itself.

In this paper, we will assume that it takes four cycles total to generate the conditional branch prediction, since we are using a 512 Kbits 2Bc-gskew predictor [14] comparable in terms of volume and accuracy with the one found on the Alpha EV8. Prediction for the branches in block F is initiated five cycles ahead, i.e it begins with only address from block A and branch history information for block Z available. One bit information per intermediate fetch block is used to compute the index of each of four tables of the 2Bcgkew predictor. This bit is different for each of the tables.

4.1.4 Branch target prediction

Targets for conditional and unconditional branches can not be computed from the information flowing from the instruction cache and must therefore be predicted. For this purpose, a branch target buffer (BTB) is used. This 16Kb BTB is indexed three cycles ahead (two cycles for reading the BTB + one cycle for final address selection). In our first experiments, a direct-mapped BTB responding in two cycles was considered. A set-associative BTB responding in three cycles was also considered. The hit ratios on these two structures were slightly disappointing due to a high number of conflict misses on the direct-mapped BTB and a larger footprint on the set-associative BTB associated with the extra access cycle.

In this paper, we report results for a tag-less 2-way skewed-associative BTB using way prediction. Different inflight information are injected in the indices of the two ways (i.e least significant bit of the cache block number on way 0 and taken or not taken transition on way 1). The way is predicted through a 16K-entries 1bit wide table. On a read on the branch target buffer, X possible targets for unconditional or conditional branches are obtained. X depends on the definition of the instruction fetch block: 1 for **0NT**, 2 for **1NT**, 3 for **1NT+** and depends on the maximum block size for **aNT**. Use of the BTB space is optimized in the following sense: any entry can be the first, the second or the Nth ($N \leq X$) target in a fetch block.

4.1.5 Return address stack

We assume that the top of return address stack associated with the two-block ahead instruction block is available, and that when the current fetch block is a call, the possible return address to be pushed on the top of stack is available. Therefore, the correct return address stack top can always be selected.

4.1.6 Jump predictor

A 16Kb jump predictor is used. The read access time of the table is two cycles. Therefore, the index used for the jump predictor hashes the address of the three-block ahead

instruction block address and the three-block ahead branch history. One bit of information per cycle (i.e, per intermediate fetch instruction block) is used in the index.

4.2 Predicting decode bits

In order to perform the address final selection, one must speculatively assert the presence/absence of conditional/unconditional branches, the presence/absence of a return or a jump, and the presence/absence of a call (and its position) in the instruction block.

4.2.1 Hardware for predicting decode bits

The decode information must be predicted the same way as the address of fetch block itself, that is (speculative) decode information must be associated with each possible block target address presented at the entry of the final IAG stage.

In order to minimize the number of decode mispredictions, we store the decode information of a target instruction block along with its address in the prediction structures, whenever possible. This stands for the BTB and for the jump predictor, but cannot be implemented for blocks in sequence and for returns.

In order to present in the decode information associated with the return target in time at the entry of final IAG stage, a (small) return decode prediction table is used. This direct-mapped table stores decode information associated with the previous return target blocks and is systematically read using the (one cycle ahead) top of the return stack as an index. Note that, in the case of chaining a call and the associated return in two successive fetch blocks, this will result in a decode misprediction[2].

In order to present the decode information associated with the sequential block A+1 after block A at the entry of the final IAG stage, we use two different techniques depending on whether block A is the target of a taken control flow instruction or not.

1) When A is the target of a taken control flow instruction the decode information for block A+1 must be presented at the entry of the final address selection at the moment when A is selected. Then there is no time to use A as an index for reading a table for getting this decode information. In order to get this decode information in time, we stored it along (when it has already been fetched once) with the decode information of block A in the different tables (if it has already been fetched once).

2) When A is a sequential block by itself A-1 the address of the preceding block was available one cycle ahead. Decode information for block A+1 is associated with the address A-1 in a table. That is the decode information associated with the two-block ahead sequential block is read on a two-block ahead decode table: this information is useful only when two successive transitions to sequential blocks occur.

[2] A few tricks can be used to limit this impact

4.2.2 Decode misprediction, but correct address generation

Occasionally, a decode misprediction does not induce directly an address generation misprediction or misfetch, for instance a not-taken conditional branch is not detected at decode prediction.

However, such a decode misprediction induces the use of faulty speculative history to read the predictions for the subsequent fetch blocks, which is very likely to result in branch mispredictions in the near term: in the example of this paper, the history length is 27. Therefore the next 27 blocks featuring conditional branches are predicted using a faulty branch history.

Therefore, a decode misprediction must be systematically repaired at decode time independently of the validity of the predicted block address: a misfetch penalty is inserted.

4.2.3 Speculative decode information width

Depending on the instruction fetch block definition, the information that must be represented in the speculative decode tables is different. The position of the last instruction in the block is needed when the block ends on a not-taken branch. It is also used to compute the address to be pushed on the return stack when the last instruction is a call.

3 bits are needed to code the type of the last control-flow information: unconditional branch, unconditional branch call, jump, jump call, return, conditional, not a control-flow instruction.

S being the maximum number of instructions in the fetch block, the widths of decode information are described below:

- **0NT**: 3 bits for the type of the last control-flow instruction in the block + $log_2 S$ bits for the position of this last instruction
- **0NT+**: same as **0NT** + 1 bit for indicating if there is an extra basic block.
- **1NT**: 1 bit ("is there a conditional branch as the first control-flow instruction?") + information for **0NT**.
- **1NT+**: same as **1NT** + 1 bit for indicating if there is an extra basic block.
- **aNT**: S bits (1 bit per instruction to represent the information "is it a conditional branch?") + 3 bits for the type of the last instruction + $log_2 S$ bits for the position of the last instruction in the fetch block.

Assuming S=16, the widths of decode information for a block are respectively 7, 8, 8, 9 and 23 bits for **0NT**, **0NT+**, **1NT**, **1NT+** and **aNT**.

4.3 Repairing on branch misprediction or an interruption

Whenever a branch misprediction or an interruption occurs, the instruction fetch must resume with the correct program counter.

A major issue with ahead pipelining the IAG is that the generation of the address of the next fetch block after the incorrectly predicted block through the normal process takes

five cycles. This would normally lead to the insertion of four bubbles in the instruction fetch pipeline. Such a solution would impair performance, by automatically lengthening the effective misprediction penalty by four cycles.

To avoid (part of) these bubble cycles, backup information may be associated with each instruction fetch block in the checkpoint/repair mechanism in order to allow to recompute (some of) the addresses of next four instruction fetch blocks on the fly without using the normal instruction address generation process.

We analyze below the information that must be stored in the checkpoint mechanism in order to allow full speed restart. Intermediate solutions suffering from one and two extra pipeline bubbles are also analyzed.

4.3.1 Predicting the next block

Let us suppose that restart occurs on block C, i.e, either the transition from block B to block C was mispredicted, or a conditional branch was predicted taken but is not-taken. Block C replaces block C'in the instruction flow.

For predicting the block D following block C on the very next cycle, one has only the time to cross the final address selection stage. First, the information needed at the entry of the final selection stage has to be directly available or immediately read from the checkpoint storage. Since only return target and sequential block address may be recomputed on the fly, target addresses provided by the BTB and the jump predictor, but also all the decode information for all possible target blocks must come from the checkpoint entry associated with block C'.

The control of the final selection must also come from the checkpoint storage. When the misprediction is associated with the direction of a conditional branch in block B or C, the speculative decode information was also available at the entry of the final address generation stage when generating the mispredicted address C': that is, it is stored in the checkpoint entry associated with block B.

When the misprediction is a jump target or a return target misprediction, the decode information associated with the fetch block was not available at the entry of the final prediction stage and must therefore be rebuilt. To rebuild the decode information, one has to wait for the instruction block coming from the instruction cache: a penalty equivalent to the one of a line misprediction penalty has to be paid.

4.3.2 Predicting the third block

If one also wants to predict block E following the block D on the second cycle then one needs all the predictions (targets, decode information, conditions) at the entry of the final selection stage by the end of the first cycle after the resuming of fetching on the corrected path. Branch targets, jump predictions and conditional branch directions can not be available in time from the IAG tables and therefore must be provided by the checkpoint storage. Since a single bit of information for each block is incorporated inflight in the indices of the tables, there are four possible sets of of these information. One of these sets is already needed in the check point storage in order to be able to recover the prediction of the first block following the mispredicted block.

Note that the access to the access to this information in the checkpoint mechanism can be pipelined on two cycles.

4.3.3 Predicting the subsequent blocks

All the possible addresses and speculative decodes can be recomputed in time for predicting blocks F and G on the subsequent cycles. But the conditional branch predictions results are not still valid in time and must be provided by the checkpoint storage.

4.3.4 Volume of information to be checkpointed

The total volume of information that must be stored in the checkpoint repair mechanism per block to allow a full speed restart is:

- branch prediction tables: a priori 2^4 x (Nb of predicted cond. branches per block) x (Nb of prediction tables in the predictor) bits are needed.
- BTB: 4 x (Nb of cond. or uncond. branches per block) addresses (+ decode information)
- jump predictor table: 4 addresses (+ decode information)
- return decode predictor: 1 decode entry
- two block ahead speculative decode table: 1 decode entry

The volume of this information is further detailed in table 1 assuming 16 bits addresses[3] and a 16 instructions maximum fetch block size and using the width of decode entries computed in Section 4.2.3. We assume that the checkpoint mechanism features two memory structures, a 1-cycle access time memory structure for storing the data needed for predicting block D and a 2-cycle access time memory structure for storing information needed for predicting the subsequent blocks. Width of the information that must be stored in the 1-cycle access time structure is represented in bold.

The volume of information to associate with each instruction fetch block in checkpoint/repair mechanism is large, especially for **aNT**. This latter case can not be realistically considered for implementation. However except for **aNT**, the total volume remains in the same range as the instruction volume by itself (512 bits). Moreover only approximately one fourth of these informations must be accessible in a single cycle.

About the delay for reading information in the checkpoint storage Reading information from the checkpoint storage is delay consuming. It should be noticed that, even in the case of using a hierarchical IAG, some information must be retrieved from the checkpoint storage (e.g. branch

[3] i.e. pointers on cache entries

	0NT	0NT+	1NT	1NT+	aNT
cond. branch pred. tables	**4** + 64	**4** + 64	**8** + 128	**8** + 128	**64** + 1024
BTB	**30** + 90	**64** + 192	**64** + 192	**102** + 306	**992** + 2976 bits
jump predictor	**30** + 90	**32** + 96	**32** + 96	**34** + 102	**62** + 186
return decode predictor	**7**	**8**	**8**	**9**	**23**
2-block ahead pred. table	**7**	**8**	**8**	**9**	**23**
total 1-cycle	**78**	**128**	**144**	**186**	**1378**
total 2-cycle	244	340	392	512	3952

Table 1. Width (in bits) of the checkpoint repair information per instruction block

history)[4]. While the information that has to be immediately read on the checkpoint storage is wider when using the ahead pipelined IAG than when using the hierarchical IAG, a 1-cycle access time 2Kb checkpoint table would be able to store a large number of inflight instruction fetch blocks except for **aNT**.

4.4 One or two bubbles restart

If the checkpoint-repair mechanism for full speed instruction fetch resuming appears to be too costly for implementation then two possible intermediate solutions exists.

First, one could insert two pipeline bubbles after the initial fetch restart cycle on block C, therefore allowing to the normal IAG tables to generate all the entries for the final multiplexor, except the conditional branch predictions. The checkpoint storage would have to provide only the set of possible conditional branch predictions.

Second, one could insert a pipeline bubble after the fetch of the first block. The checkpoint storage could then provide all the entries that must be presented to the final address selection to predict the second block (block D), but using a 2-cycle access time table. The tables of the ahead pipelined IAG can provide in time all the entries for the final selection for the third and fourth fetch block (block E and F) apart conditional branch predictions.

5 Evaluation framework

In order to evaluate ahead pipelined IAG against usual hierarchical IAG, we run trace driven simulations.

5.1 Benchmark selection and description

Traces were recorded on the SPEC2000 benchmarks (11 int and 6 floats) on a Solaris Sun workstations. Program were compiled with the Sun compiler with optimization -xO5 in order to avoid the artifacts associated with non-optimized codes. Reference inputs were used. 100 millions instructions were collected after skipping the initialization phases of the programs.

5.2 Metrics

In our evaluation, we use *misp/KI* and *misfetches/KI*, the average numbers of mispredictions (resp. misfetches) per kilo-instructions. *misp/KI* allows to characterize the accuracy of the overall address prediction executed by the IAG. *misfetches/KI* allows to characterize the quality of the "in-time" address prediction of the IAG.

The goal of ahead pipelining the IAG is to increase the instruction fetch bandwidth through substantially reducing *misfetches/KI* while keeping *misp/KI* in the same range.

5.3 About predictor tables update

The simulation results reported in this paper implicitly assume immediate predictor update on the right path, and consider that no update on the predictor tables is performed for instructions on the wrong paths.

In a real machine, the effective update is performed later in the pipeline, either after the branch misfetch or misprediction resolution or at commit time. The conditional branch predictor and the jump predictor must wait for misprediction resolution for updating. However, the effective accuracy of the conditional branch predictor (resp. a jump predictor) updated at commit time or misprediction resolution time would not be significantly different from the ones reported in this paper. The predictions for an instruction-history pair (I,H) could only be different if this prediction shares an entry in the table with a former mispredicted instruction J which has been already executed but is still not committed. Such cases occur either on interferences in a table or if I and J have same address and the same history and path. As we use long histories and large tables, both these phenomena occur very rarely. The impact on prediction accuracy of these phenomena was independently reported as neglectable in [12] and in [6].

The other tables, respectively the line predictor for the hierarchical IAG and the BTB, the return decode predictor table and the two block-ahead decode table for the ahead pipelined IAG, can be updated at misfetch time, resolution time or commit time. Updating them only at commit time or execution time would induce a lot of intermediate misfetches. For instance, if a misfetch is encountered on the first iteration, then successive misfetches on the succeeding iterations will be encountered till first iteration commits. Our simulation assumes update at misfetch resolution time, but ignores all the impacts of fetching of the wrong paths, i.e.,

[4]This information has also to be immediately retrieved on the ahead pipelined IAG

pollution (an entry is spared by blocks that will not be executed on the right path), prefetch (a misfetch occurs on the wrong path on a block that will be executed on the right path), and prediction destruction or correction (an entry previously set by a block B is modified by same block B: this may result either in an extra misfetch or avoid a misfetch). The two former phenomena occur for both the hierarchical IAG and the ahead pipelined IAG. The latter phenomenon occurs only for the hierarchical IAG on the line predictor.

5.4 Simulated configurations

The principal parameters of the simulated configurations for both ahead pipelined IAG and the conventional IAG are recalled below. Except for the branch predictor, the following rule was applied: 16 Kb size for tables with 2-cycle read access time and 2Kb size for tables with 1-cycle read access time.

- 512 Kbits 2Bc-gskew conditional branch predictor. 3 cycles read for the base design, 4 cycles for ahead pipelined design. A single bit of history per fetch block is inserted (as on EV8 [12]). History lengths are respectively: 0, 17, 27 and 20 for 4 tables in the predictor. No specific study was done to chose the best combination of history lengths since it might vary for each fetch instruction block definition.
- 16Kb jump predictor, 2 cycles read, 12 history bits are hashed with the available address. 8Kentries on the base design and 4Kentries on the ahead pipelined design.
- return stack: 32 entries
- ahead pipelined IAG: a 1Kentries 1-cycle access time return decode, a 2Kentries 1-cycle read two-block ahead decode table and a 4Kentries 2-cycle read BTB.

- Base design: a 1Kentries line predictor (including a 4K 1-bit direction prediction)

As mentioned in Section 4.1.1, inflight information is injected during the instruction address generation. In the index of a table, the injected information is one bit wide per intermediate fetch block: when a predictor features several tables or several ways, different information are injected to index the different ways. The inflight informations used in the results reported in this paper are combinations (by exclusive-OR) between the least significant bits in the address of the block and the information "the block is in sequence with the previous one", This inflight information is injected for the hierarchy of IAGs as well as for the ahead pipelined IAG as illustrated in Figures 2 and 3.

6 Performance evaluation

In this section, we first describe the average fetch block size for the different definitions of the fetch block. We then compare the accuracies of ahead pipelined IAG and hierarchy of IAGs; ahead pipelined IAG is slightly less accurate than the hierarchy of IAGs. Then we show that the hierarchy of IAG wastes a significant part of the instruction address generation bandwidth in pipeline bubbles when the fast and slow IAGs disagree. The ahead pipelined IAG is shown to allow an instruction address generation throughput close to one instruction block throughput.

6.1 Average fetch width

Tables 2 and 3 presents the average sizes of dynamic basic blocks (sequences of contiguous instructions ending by a control flow instruction) and instruction streams (sequences of contiguous instructions ending by a taken control flow instruction [9]) on our set of benchmarks. The average maximum number of contiguous instructions that can be fetched in parallel generally stands between 7 and 12 on integer applications and is significantly higher on some floating point applications.

Table 4 presents the average size of the fetch block sizes for the different instruction block fetch definitions on our our set of integer benchmarks[5]. One can remark that, apart when considering a cache line size of 16 instructions and the **Rl** policy (i.e being able to fetch up to 32 instructions per cycle), the potential bandwidth advantage enabled by allowing to bypass every not-taken branch (**aNT**) instead of bypassing a single not-taken branch (**1NT+**) remains very limited (less than 2 %). On the other hand, extending the fetch block ending condition from **0nt** to **1NT+** is much more (potentially) beneficial: from 9% fetch width increase for **Rs** and a 4-instruction cache line to 31 % fetch width increase for **Rl** and a 16-instruction cache line. As it could be expected, the average fetch block size on floating-point applications (not represented here) is wider than on integer applications. Relaxing constraints on the block range and on bypassing not-taken branches also enlarge the average block size, but in a more limited amplitude than on integer benchmarks.

6.2 Accuracy of the IAG prediction

Figures 5 and 6 present the average accuracies in *misp/KIs* for the two IAGs (hier. for hierarchy of IAGs, for ahead pipelined IAG) for the whole spectrum of instruction fetch block definitions for a cache line sizes varying from 4 to 16 instructions. For space reasons, we only illustrate results for the **Rl** policy, similar trends are encountered for **Rs** and **Rc** policies.

With the integer applications, the accuracy of the two IAGs remains in the same range with generally a small advantage for the hierarchical IAG. For all benchmarks and all policies, the results reflected the same trend: very close accuracies on both IAGs for the whole spectrum of instruction fetch blocks, sometimes in favor of ahead pipelined IAG slightly more often in favor of the hierarchy of IAGs.

[5]For space reasons, we present the average on all benchmarks. The presented statistic is the average of the average size, i.e $\frac{1}{n}\sum\frac{Nb\ Inst}{Nb\ Blocks}$.

	gzip	vpr	gcc	mcf	craf	pars	vort	bzip2	twol	gap	perl	**avg**
inst/basic block	7.62	6.18	4.91	4.72	6.98	5.22	5.64	5.12	8.60	12.21	6.09	**6.66**
inst/inst. stream	10.53	8.94	7.22	7.93	10.90	8.08	9.08	6.85	11.82	18.80	7.55	**9.79**

Table 2. Average size of basic blocks and instruction streams on integer benchmarks

	wupwise	equake	facerec	fma3d	ammp	apsi	**avg**
inst/basic block	20.42	10.83	9.02	5.19	12.82	35.59	**15.64**
inst/inst. stream	23.83	12.82	11.67	7.92	16.86	39.24	**18.72**

Table 3. Average size of basic blocks and instruction streams on floating point benchmarks

With floating point applications, the similarities of the accuracies is even more pronounced.

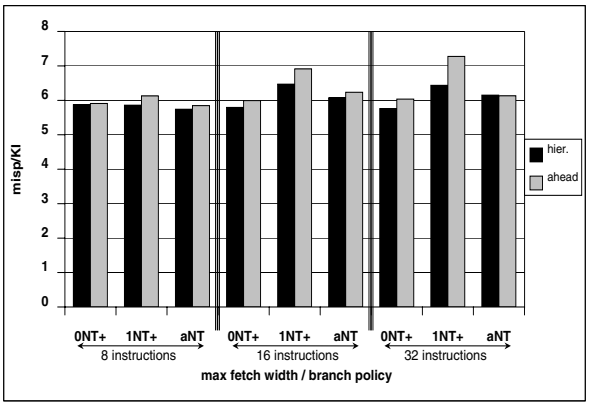

Figure 5. Average accuracies of the two IAGs (misp/KI) on integer applications

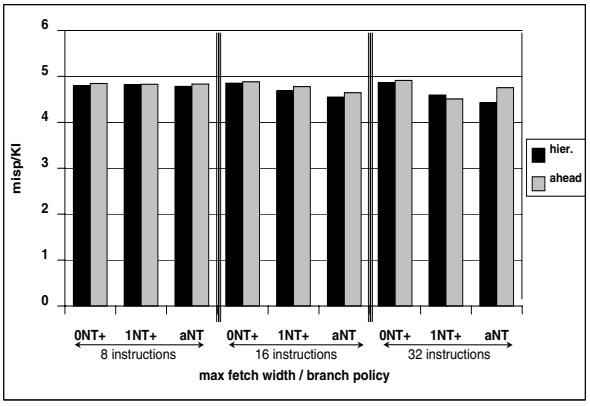

Figure 6. Average accuracies of the two IAGs (misp/KI) on floating-point applications

6.3 Line predictor versus ahead pipelined IAG decode and target predictor

Line misprediction (but not IAG misprediction) on the hierarchy of IAGs as well as decode or branch target misprediction on the ahead pipelined IAG result in a misfetch (i.e address generation is corrected at decode time). A misfetch induces a three cycle IAG penalty.

In Section 3, we pointed out that the size of a line predictor responding in a single cycle is limited by technological constraints. Nevertheless, our experiment showed that considering a small 1Kentry (i.e. 2Kb) single cycle line predictor is generally more effective than a larger (but two block ahead) line predictor. This is associated with the use of more paths for single address block on a two-block ahead line predictor.

Figures 7 and 8 illustrate the respective average fetch accuracy of the line predictor and of the ahead pipelined IAG. On integer applications, line prediction performs very poorly and a significant number of fetch cycles are wasted on each correction of the line prediction by the slow IAG. Better behavior is encountered on floating-point applications (the line predictor is able to capture regular behavior). On the other hand, decode and target prediction is quite effective on the ahead pipelined IAG: in average the ahead pipelined IAG generates about 5 to 10 times less misfetches than the hierarchy of IAGs.

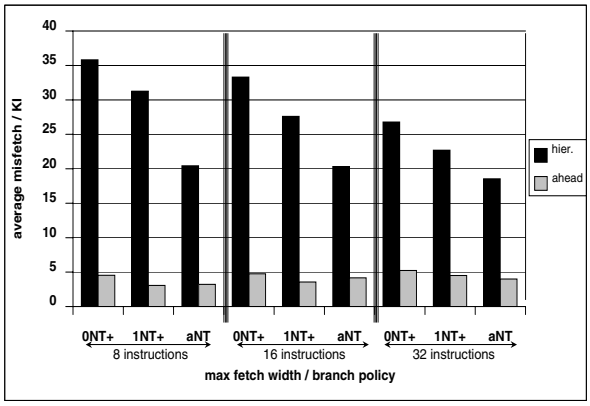

Figure 7. Average misfetches/KI (integer applications)

The impact of this phenomenon on the overall instruction address generation bandwidth for integer benchmarks is illustrated on Figure 9. In this figure, we report MIFC, the

cache line width	4			8			16		
	Rs	**Rc**	**Rl**	**Rs**	**Rc**	**Rl**	**Rs**	**Rc**	**Rl**
0nt	2.745	3.245	4.583	3.860	4.840	5.813	4.851	5.987	6.363
0NT	2.971	3.366	4.765	4.225	4.961	5.919	5.159	6.073	6.402
0NT+	3.035	3.399	5.054	4.415	5.292	6.547	5.569	6.805	7.231
1nt	3.046	3.414	5.239	4.561	5.610	7.188	5.749	7.569	8.175
1NT	3.048	3.414	5.261	4.610	5.613	7.203	6.053	7.590	8.182
1NT+	3.048	3.414	5.272	4.620	5.637	7.288	6.112	7.768	8.383
aNT	3.048	3.414	5.275	4.625	5.640	7.401	6.199	7.924	8.801
Max fetch width	4			8			16		32

Table 4. Average fetch block sizes on integer benchmarks

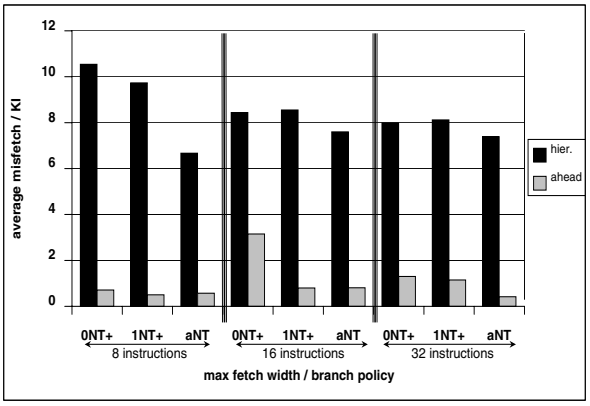

Figure 8. Average misfetches/KI (floating point applications)

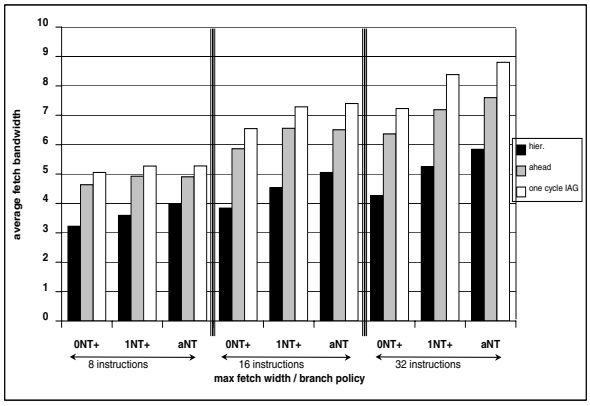

Figure 9. Average instruction fetch bandwidth on the correct path

average maximum instruction fetch bandwidth on the correct path for a hierarchy of IAGs and an ahead pipelined IAG and compare it to the average fetch block size that would be achieved by a single cycle IAG. This figure clearly shows that line misprediction spares about one third of the potential IAG bandwidth with integer applications. This loss of IAG bandwidth explains why the Alpha EV8 was implementing the fetch of two blocks per cycle. On the other hand, the loss of bandwidth associated with misfetches when using ahead pipelined IAG is much more limited (around 10 %). With floating point applications, the respective IAG bandwidth loss (not illustrated) are in the 8-12 % range for the hierarchy of IAGs and the 1-5 % range for the ahead pipelined IAG.

6.4 Potential impact of the overall performance

The simulation results reported so far can be summarized as follows. With integer applications, a hierarchy of IAGs is slightly more accurate an ahead pipelined IAG, but an ahead pipelined IAG enables a significantly higher instruction fetch throughput. The impact on effective performance of a processor would depend on many other architectural parameters.

However the number of cycles lost due to an imperfect IAG can be roughly modeled by the following formula: $nbmisfetch * misfetchpenalty + nbmisprediction * mispredictionpenalty$.

Table 5 reports the misprediction penalty threshold above which the hierarchy of IAGs would lose less cycles in average than the ahead pipelined IAG in integer benchmarks[6]. The very high values of these thresholds clearly illustrate that the slight loss of accuracy of ahead pipelined IAG will be more than compensated by the significant instruction fetch bandwidth increase.

7 Summary and future works

An instruction address generator using a combination of state-of-the-art accurate predictors cannot respond in a single cycle. The use of a hierarchy of IAGs (e.g. line predictors + state-of-the-art IAG) leads to the waste of a significant part of the instruction fetch bandwidth.

In this paper, we have explored in details an alternative to the use of a hierarchy of IAGs, ahead pipelining the IAG. The effective instruction fetch bandwidth enabled by our proposal is significantly higher than its counterpart with

[6]When ahead prediction is more accurate than conventional prediction, ahead pipelined IAG loses less cycles for all possible penalties.

cache line width	4			8			16		
	Rs	**Rc**	**Rl**	**Rs**	**Rc**	**Rl**	**Rs**	**Rc**	**Rl**
0NT+	431	INF	2841	667	6922	431	509	570	233
1NT+	443	INF	313	444	668	162	132	203	64
aNT	286	424	480	4427	696	303	INF	399	INF
Max fetch width	4			8			16		32

Table 5. Misprediction penalty threshold (cycles)

using a conventional hierarchy of IAGs. Maintaining the accuracy is enabled by the use of inflight information that were not already available at the initiation of the prediction. This is particularly true for conditional branch prediction, where the initiation of the table reads five cycles ahead has very limited impact on the accuracy. Ahead pipelining of the IAG results in a gain in instruction address generation throughput only if one is able to correctly assert in time the decode information needed to control the IAG. We have presented a complete decode information predictor intimately inserted in the IAG. This decode predictor is also ahead pipelined and achieves high accuracy, at least compared with line prediction. After a misprediction, the ahead pipelined AHIAG allows smooth instruction fetch resuming without extra misprediction penalty, but at the extra cost of saving additional information in the checkpoint/repair mechanism.

The instruction fetch throughput allowed by the IAG presented in this paper seems sufficient to feed the execution core of 4-to-6 way issue processors, but may fail a little bit short for 8-way (or higher) issue processors on branch intensive applications. Code layout techniques favoring not-taken branches may substantially improve the effective address generation bandwidth [9]. In order to further increase the instruction fetch bandwidth, we will study the adaptation of ahead pipelining IAGs to a large spectrum of instruction fetch front ends with increasing complexity: decoupled instruction fetch front end [10], multiple (non-contiguous) block address generation [16, 2, 13, 15] and trace caches [4, 11].

References

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

[2] Simonjit Dutta and Manoj Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Ann Arbor, Michigan, 1995.

[3] J. Emer. Ev8, the postultimate alpha. In *presentation at PACT' 2001, Barcelona*, 2001.

[4] Daniel Holmes Friendly, Sanjay Jeram Patel, and Yale N. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 24–33, Research Triangle Park, North Carolina, December 1–3, 1997. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[5] Linley Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 1995.

[6] D. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th International Symposium on High Perform ance Computer Architecture*, 2003.

[7] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[8] Pierre Michaud, André Seznec, and Stéphan Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *International Journal of Parallel Programming*, Feb 2001.

[9] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert Cohn, Josep Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[10] G. Reinman, B. Calder, and T. Austin. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, 1999.

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

[12] André Seznec, Stephen Felix, Venkata Krishnan, and Yanos Sazeidès. Design tradeoffs for the ev8 branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[13] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Proceedings of the 7th conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[14] André Seznec and Pierre Michaud. De-aliased hybrid branch predictors. Technical Report RR-3618, Inria, Institut National de Recherche en Informatique et en Automatique, 1999.

[15] Steven Wallace and Nader Bagherzadeh. Multiple branch and block prediction. In *Proceedings of 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97)*, 1997.

[16] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proc. of ACM 1993 Int. Conf. on Supercomputing (7th ICS'93)*, 1993.