Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor*

André Seznec
IRISA/INRIA
Campus de Beaulieu
35042 Rennes
France
seznec@irisa.fr

Stephen Felix
Intel
334 South Street
Shrewsbury, MA 01545
USA
Stephen.Felix@intel.com

Venkata Krishnan
StarGen, Inc.
225 Cedar Hill Street
Marlborough, MA 01752
USA
krishnan@stargen.com

Yiannakis Sazeides
Dept of Computer Science
University of Cyprus
CY-1678 Nicosia
Cyprus
yanos@cs.ucy.ac.cy

Abstract

This paper presents the Alpha EV8 conditional branch predictor. The Alpha EV8 microprocessor project, canceled in June 2001 in a late phase of development, envisioned an aggressive 8-wide issue out-of-order superscalar microarchitecture featuring a very deep pipeline and simultaneous multithreading. Performance of such a processor is highly dependent on the accuracy of its branch predictor and consequently a very large silicon area was devoted to branch prediction on EV8. The Alpha EV8 branch predictor relies on global history and features a total of 352 Kbits.

The focus of this paper is on the different trade-offs performed to overcome various implementation constraints for the EV8 branch predictor. One such instance is the pipelining of the predictor on two cycles to facilitate the prediction of up to 16 branches per cycle from any two dynamically successive, 8 instruction fetch blocks. This resulted in the use of three fetch-block old compressed branch history information for accesing the predictor. Implementation constraints also restricted the composition of the index functions for the predictor and forced the usage of only single-ported memory cells.

Nevertheless, we show that the Alpha EV8 branch predictor achieves prediction accuracy in the same range as the state-of-the-art academic global history branch predictors that do not consider implementation constraints in great detail.

1 Introduction

The Alpha EV8 microprocessor [2] features a 8-wide superscalar deeply pipelined microarchitecture. With minimum branch misprediction penalty of 14 cycles, the performance of this microprocessor is very dependent on the branch prediction accuracy. The architecture and technology of the Alpha EV8 are very aggressive and new challenges were confronted in the design of the branch predictor. This paper presents the Alpha EV8 branch predictor in great detail. The paper expounds on different constraints that were faced during the definition of the predictor, and on various trade-offs performed that lead to the final design. In particular, we elucidate on the following: (a) use of a global history branch prediction scheme, (b) choice of the prediction scheme derived from the hybrid skewed branch predictor *2Bc-gskew*[19], (c) redefinition of the information vector used for indexing the predictor that combines compressed branch history and path history, (d) different prediction and hysteresis table sizes: prediction tables and hysteresis tables are accessed at different pipeline stages, and hence can be implemented as physically distinct tables, (e) variable history lengths: the four logical tables in the EV8 predictor are accessed using four different history lengths, (f) guaranteeing conflict free access to the bank-interleaved predictor with single-ported memory cells for up to 16 branch predictions from any two 8-instruction dynamically succesive fetch blocks, and (g) careful definition of index functions for the predictor tables.

This work demonstrates that in spite of all the hardware and implementation constraints that were encountered, the Alpha EV8 branch predictor accuracy was not compromised and stands the comparison with virtually all equivalent in size, global history branch predictors that have been proposed so far.

The overall EV8 architecture was optimized for single process performance. Extra performance obtained by simultaneous multithreading was considered as a bonus. Therefore, the parameters of the conditional branch predictor were tuned with single process performance as the primary objective. However, the EV8 branch predictor was found to perform well in the presence of a multithreaded workload.

The remainder of the paper is organized as follows. Section 2 briefly presents the instruction fetch pipeline of the Alpha EV8. Section 3 explains why a global history branch predictor scheme was preferred over a local. In Section 4, we present the prediction scheme implemented in the Alpha EV8, *2Bc-gskew*. This section also presents the design space of *2Bc-gskew*. The various design dimensions were

*This work was done while the authors were with Compaq during 1999

harnessed to fit the EV8 predictor in 352 Kbits memory budget. Section 5 presents and justifies the history and path information used to index the branch predictor. On the Alpha EV8, the branch predictor tables must support two independent reads of 8 predictions per cycle. Section 6 presents the scheme used to guarantee two conflict-free accesses per cycle on a bank-interleaved predictor. Section 7 presents the hardware constraints for composing index functions for the prediction tables and describes the functions that were eventually used. Section 8 presents a step by step performance evaluation of the EV8 branch predictor as constraints are added and turn-around solutions are adopted. Finally, we provide concluding remarks in Section 9.

2 Alpha EV8 front-end pipeline

To sustain high performance, the Alpha EV8 fetches up to two, 8-instruction blocks per cycle from the instruction cache. An instruction fetch block consists of all consecutive valid instructions fetched from the I-cache: an instruction fetch block ends either at the end of an aligned 8-instruction block or on a **taken** control flow instruction. **Not taken** conditional branches do not end a fetch block, thus up to 16 conditional branches may be fetched and predicted in every cycle.

On every cycle, the addresses of the next two fetch blocks must be generated. Since this must be achieved in a single cycle, it can only involve very fast hardware. On the Alpha EV8, a *line predictor* [1] is used for this purpose. The line predictor consists of three tables indexed with the address of the most recent fetch block and a very limited hashing logic. A consequence of simple indexing logic is relatively low line prediction accuracy.

To avoid huge performance loss, due to fairly poor line predictor accuracy and long branch resolution latency (on the EV8 pipeline, the outcome of a branch is known the earliest in cycle 14 and more often around cycle 20 or 25), the line predictor is backed up with a powerful program counter (PC) address generator. This includes a conditional branch predictor, a jump predictor, a return address stack predictor, conditional branch target address computation (from instructions flowing out of the instruction cache) and final-address selection. PC-address-generation is pipelined in two cycles as illustrated in Fig. 1: up to four dynamically succesive fetch blocks A, B, C and D are simultaneously in flight in the PC-address-generator. In case of a mismatch between line prediction and PC-address-generation, the instruction fetch is resumed with the PC-address-generation result.

3 Global vs Local history

The previous generation Alpha microprocessor [7] incorporated a hybrid predictor using both global and local branch history information. On Alpha EV8, up to 16 branch outcomes (8 for each fetch block) have to be predicted per

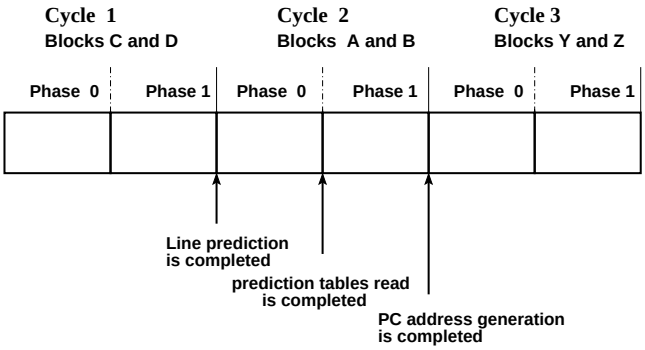

Figure 1. PC address generation pipeline

cycle. Implementing a hybrid branch predictor for EV8 based on local history or including a component using local history would have been a challenge.

Local branch prediction requires for each prediction a read of the local history table and then a read of the prediction table. Performing the 16 local history reads in parallel requires a dual-ported history table. One port for each fetch block is sufficient since one can read in parallel the histories for sequential instructions on sequential table entries. But performing the 16 prediction table reads would require a 16-ported prediction table.

Whenever an occurrence of a branch is inflight, the speculative history associated with the younger inflight occurrence of the branch should be used [8]. Maintaining and using speculative local history is already quite complex on a processor fetching and predicting a single branch per cycle[20]. On Alpha EV8, the number of inflight branches is possibly equal to the maximum number of inflight instructions (that is more than 256). Moreover, in EV8 when indexing the branch predictor there are up to three fetch blocks for which the (speculative) branch outcomes have not been determined (see Fig. 1). These three blocks may contain up to three previous occurrences of every branch in the fetch block. In contrast, single speculative global history (per thread) is simpler to build and as shown in Section 8 the accuracy of the EV8 global history prediction scheme is virtually insensitive to the effects of three fetch blocks old global history.

Finally, the Alpha EV8 is a simultaneous multithreaded processor [25, 26]. When independent threads are running, they compete for predictor table entries. Such interference on a local history based scheme can be disastrous, because it pollutes both the local history and prediction tables. What is more, when several parallel threads are spawned by a single application, the pollution is exacerbated unless the local history table is indexed using PC **and** thread number. In comparison, for global history schemes a global history register must be maintained per thread, and parallel threads - from the same application - benefit from constructive aliasing [10].

4 The branch prediction scheme

Global branch history branch predictor tables lead to a phenomenon known as *aliasing* or *interference* [28, 24], in which multiple branch information vectors share the same entry in the predictor table, causing the predictions for two or more branch substreams to intermingle. "De-aliased" global history branch predictors have been recently introduced: the enhanced skewed branch predictor *e-gskew* [15], the agree predictor [22], the bimode predictor [13] and the YAGS predictor [4]. These predictors have been shown to achieve higher prediction accuracy at equivalent hardware complexity than larger "aliased" global history branch predictors such as *gshare* [14] or *GAs* [27]. However, hybrid predictors combining a global history predictor and a typical bimodal predictor only indexed with the PC [21] may deliver higher prediction accuracy than a conventional single branch predictor [14]. Therefore, "de-aliased" branch predictors should be included in hybrid predictors to build efficient branch predictors.

The EV8 branch predictor is derived from the hybrid skewed branch predictor *2Bc-gskew* presented in [19]. In this section, the structure of the hybrid skewed branch predictor is first recalled. Then we outline the update policy used on the EV8 branch predictor. The three degrees of freedom available in the design space of the *2Bc-gskew* predictor are described: different history lengths for the predictor components, size of the different predictor components and using smaller hysteresis tables than prediction tables. These degrees of freedom were leveraged to design the "best" possible branch predictor fitting in the EV8 hardware budget constraints.

4.1 General structure of the hybrid skewed predictor 2Bc-gskew

The enhanced skewed branch predictor *e-gskew* is a very efficient single component branch predictor [15, 13] and therefore a natural candidate as a component for a hybrid predictor. The hybrid predictor *2Bc-gskew* illustrated in Fig. 2 combines *e-gskew* and a bimodal predictor. *2Bc-gskew* consists of four 2-bit counters banks. Bank **BIM** is the bimodal predictor, but is also part of the *e-gskew* predictor. Banks **G0** and **G1** are the two other banks of the *e-gskew* predictor. Bank **Meta** is the meta-predictor. Depending on **Meta**, the prediction is either the prediction coming out from **BIM** or the majority vote on the predictions coming out from **G0**, **G1** and **BIM**

4.2 Partial update policy

In a multiple table branch predictor, the update policy can have a bearing on the prediction accuracy [15]. Partial update policy was shown to result in higher prediction accuracy than total update policy for *e-gskew*.

Applying partial update policy on *2Bc-gskew* also results in better prediction accuracy. The bimodal component accurately predicts strongly biased static branches. Therefore, once the metapredictor has recognized this situation, the other tables are not updated and do not suffer from aliasing associated with easy-to-predict branches.

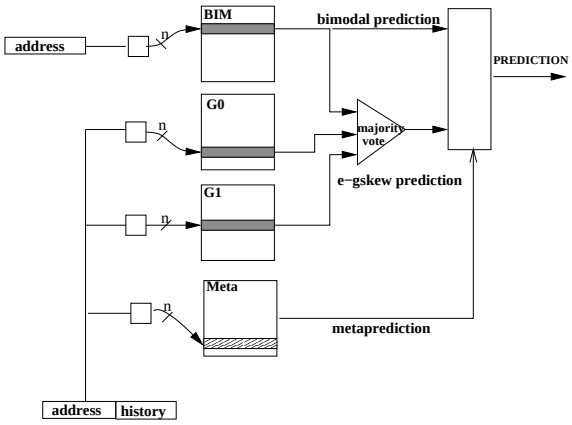

Figure 2. The *2Bc-gskew* predictor

The partial update policy implemented on the Alpha EV8 consists of the following:

- **on a correct prediction**:
  when all predictors were agreeing do not update (see Rationale 1)
  otherwise: strengthen **Meta** if the two predictions were different, and strengthen the correct prediction on all participating tables **G0**, **G1** and **BIM** as follows:
  -strengthen **BIM** if the bimodal prediction was used
  -strengthen all the banks that gave the correct prediction if the majority vote was used
- **on a misprediction**:
  when the two predictions were different, first update the chooser (see Rationale 2), then recompute the overall prediction according to the new value of the chooser:
  -correct prediction: strengthens all participating tables
  -misprediction: update all banks

Rationale 1 The goal is to limit the number of strengthened counters on a correct prediction. When a counter is strengthened, it is harder for another (address,history) pair to "steal" it. But, when the three predictors **BIM**, **G0** and **G1** are agreeing, one counter entry can be stolen by another (address, history) pair without destroying the majority prediction. By not strengthening the counters when the three predictors agree, such a stealing is made easier.

Rationale 2 The goal is to limit the number of counters written on a wrong prediction: there is no need to steal a table entry from another (address, history) pair when it can be avoided.

4.3 Using distinct prediction and hysteresis arrays

Partial update leads to better prediction accuracy than total update policy due to better space utilization. It also

allows a simpler hardware implementation of a hybrid predictor with 2-bit counters.

When using the partial update described earlier, on a correct prediction, the prediction bit is left unchanged (and not written), while the hysteresis bit is strengthened on participating components (and need not be read). Therefore, a correct prediction requires only one read of the prediction array (at fetch time) and (at most) one write of the hysteresis array (at commit time). A misprediction leads to a read of the hysteresis array followed by possible updates of the prediction and hysteresis arrays.

4.4 Sharing a hysteresis bit between several counters

Using partial update naturally leads to a *physical* implementation of the branch predictor as two different memory arrays, a prediction array and a hysteresis array.

For the Alpha EV8, silicon area and chip layout constraints allowed less space for the hysteresis memory array than the prediction memory array. Instead of reducing the size of the prediction array, it was decided to use half size hysteresis tables for components **G1** and **Meta**. As a result, two prediction entries share a single hysteresis entry: the prediction table and the hysteresis table are indexed using the same index function, except the most significant bit.

Consequently, the hysteresis table suffers from more aliasing than the prediction table. For instance, the following scenario may occur. Prediction entries A and B share the same hysteresis entry. Both (address, history) pairs associated with the entries are strongly biased, but B remains always wrong due to continuous resetting of the hysteresis bit by (address, history) pair associated with A. While such a scenario certainly occurs, it is very rare: any two consecutive accesses to B without intermediate access to A will allow B to reach the correct state. Moreover, the partial update policy implemented on the EV8 branch predictor limits the number of writes on the hysteresis tables and therefore decreases the impact of aliasing on the hysteresis tables.

4.5 History lengths

Previous studies of the skewed branch predictor [15] and the hybrid skewed branch predictor [19] assumed that tables **G0** and **G1** were indexed using different hashing function on the (address, history) pair but with the same history length used for all the tables. Using different history lengths for the two tables allows slightly better behavior. Moreover as pointed out by Juan et al. [12], the optimal history length for a predictor varies depending on the application. This phenomenon is less important on a hybrid predictor featuring a bimodal table as a component. Its significance is further reduced on *2Bc-gskew* if two different history lengths are used for tables **G0** and **G1**. A medium history length can be used for **G0** while a longer history length is used for **G1**.

	BIM	G0	G1	Meta
prediction table	16K	64K	64K	64K
hysteresis table	16K	32K	64K	32K
history length	4	13	21	15

Table 1. Characteristics of Alpha EV8 branch predictor

4.6 Different prediction table sizes

In most academic studies of multiple table predictors [15, 13, 14, 19], the sizes of the predictor tables are considered equal. This is convenient for comparing different prediction schemes. However, for the design of a real predictor in hardware, the overall design space has to be explored. Equal table sizes in the *2Bc-gkew* branch predictor is a good trade-off for small size predictors (for instance 4*4K entries). However, for very large branch predictors (i.e 4 * 64K entries), the bimodal table **BIM** is used very sparsely since each branch instruction maps onto a single entry.

Consequently, the large branch predictor used in EV8 implements a **BIM** table smaller than the other three components.

4.7 The EV8 branch predictor configuration

The Alpha EV8 implements a very large *2Bc-gskew* predictor. It features a total of 352 Kbits of memory, consisting of 208 Kbits for prediction and 144 Kbits for hysteresis. Design space exploration lead to the table sizes indexed with different history lengths as listed in Table 1. It may be remarked that the table BIM (originally the bimodal table) is indexed using a 4-bit history length. This will be justified when implementation constraints are discussed in Section 7.

5 Path and branch outcome information

The accuracy of a branch predictor depends both on the prediction scheme and predictor table sizes as well as on the information vector used to index it. This section describes how pipeline constraints lead to the effective information vector used for indexing the EV8 Alpha branch predictor. This information vector combines the PC address, a compressed form of the three fetch blocks old *branch and path* history and path information form the three last blocks.

5.1 Three fetch blocks old block compressed history

Three fetch blocks old history Information used to read the predictor tables must be available at indexing time. On the Alpha EV8, the branch predictor has a latency of two cycles and two blocks are fetched every cycle. Fig. 1 shows that the branch history information used to predict

a branch outcome in block D can not include any (speculative) branch outcome from conditional branches in block D itself, and also from blocks C, B and A. Thus the EV8 branch predictor can only be indexed using a three fetch blocks old branch history (i.e updated with history information from Z) for predicting branches in block D.

Block compressed history lghist When a single branch is predicted per cycle, at most one history bit has to be shifted in the global history register on every cycle. When up to 16 branches are predicted per cycle, up to 16 history bits have to be shifted in the history on every cycle. Such an update requires complex circuitry. On the Alpha EV8, this complex history-register update would have stressed critical paths to the extent that even older history would have had to be used (five or even seven-blocks old).

Instead, just a single history bit is inserted per fetch block [5]. The inserted bit combines the last branch outcome with path information. It is computed as follows: whenever at least one conditional branch is present in the fetch block, the outcome of the last conditional branch in the fetch block (1 for taken, 0 for not-taken) is exclusive-ORed with bit 4 in the PC address of this last branch. The rationale for exclusive-OR by a PC bit the branch outcome is to get a more uniform distribution of history patterns for an application. Highly optimized codes tend to exhibit less taken branches than not-taken branches. Therefore, the distribution of "pure" branch history outcomes in those applications is non-uniform.

While using a single history bit was originally thought of as a compromising design trade-off - since it is possible to compress up to 8 history bits into 1 - Section 8 shows that it does not have significant effect on the accuracy of the branch predictor.

Notation The block compressed history defined above will be referred to as *lghist*.

5.2 Path information from the three last fetch blocks

Due to EV8 pipeline constraints (Section 2), three fetch-blocks old *lghist* is used for the predictor. Although, no branch history information from these three blocks can be used, their addresses are available for indexing the branch predictor. The addresses of the three previous fetch blocks are used in the index functions of the predictor tables.

5.3 Using very long history

The Alpha EV8 features a very large branch predictor compared to those implemented in previous generation microprocessors. Most academic studies on global history branch predictors have assumed that the length of the global history is smaller or equal to log_2 of the number of entries of the branch predictor table. For the size of predictor used in Alpha EV8, this is far from optimal even when using *lghist*. For example, when considering "not compressed" branch history for a 4*64K 2-bit entries *2Bc-gskew* predictor, using equal history length for **G0**, **G1** and **Meta**, history length 24 was found to be a good design point. When considering different history lengths, using 17 for **G0**, 20 for **Meta** and 27 for **G1** was found to be a good trade-off.

For the same predictor configuration with three fetch blocks old *lghist*, slightly shorter length was found to be the best performing. However, the optimal history length is still longer than log_2 of the size of the branch predictor table: for the EV8 branch predictor 21 bits of *lghist* history are used to index table G1 with 64K entries.

In Section 8, we show empirically that for large predictors, branch history longer than log_2 of the predictor table size is almost always beneficial.

6 Conflict free bank interleaved branch predictor

Up to 16 branch predictions from two fetch blocks must be computed in parallel on the Alpha EV8. Normally, since the addresses of the two fetch blocks are independent, each of the branch predictor tables would have had to support two independent reads per cycle. Therefore the predictor tables would have had to be multi-ported, dual-pumped or bank-interleaved. This section presents a scheme that allowed the implementation of the EV8 branch predictor as 4-way bank interleaved using only single-ported memory cells. Bank conflicts are avoided **by construction**: the predictions associated with two dynamically successive fetch blocks are assured to lie in two distinct banks in the predictors.

6.1 Parallel access to predictions associated with a single block

Parallel access to all the predictions associated with a single fetch block is straightforward. The prediction tables in the Alpha EV8 branch predictor are indexed based on a hashing function of address, three fetch blocks old *lghist* branch and path history, and the three last fetch block addresses. For all the elements of a single fetch block, the same vector of information (except bits 2, 3 and 4 of the PC address) is used. Therefore, the indexing functions used guarantee that eight predictions lie in a single 8-bit word in the tables.

6.2 Guaranteeing two successive non-conflicting accesses

The Alpha EV8 branch predictor must be capable of delivering predictions associated with two fetch blocks per clock cycle. This typically means the branch predictor must be multi-ported, dual-pumped or bank interleaved.

On the Alpha EV8 branch predictor, this difficulty is circumvented through a bank number computation. The bank number computation described below guarantees **by construction** that any two dynamically successive fetch

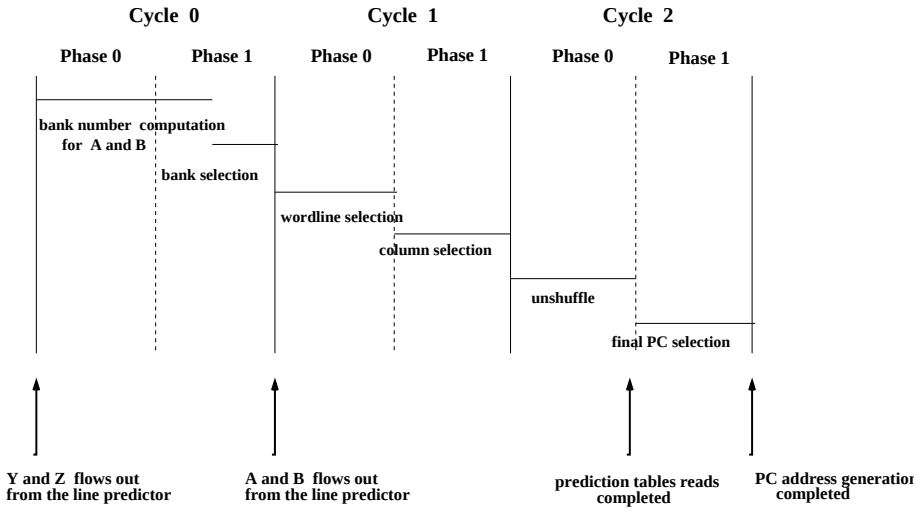

Figure 3. Flow of the branch predictor tables read access

blocks will generate accesses to two distinct predictor banks. Therefore, bank conflicts never occur. Moreover, the bank number is computed on the same cycle as the address of the fetch block is generated by the line predictor, thus no extra delay is added to access the branch predictor tables (Fig. 3). The implementation of the bank number computation is defined below:

let B_A be the bank number for instruction fetch block A, let Y, Z be the addresses of the two previous access slots, let B_Z be the number of the bank accessed by instruction fetch block Z, let (y52,y51,..,y6,y5,y4,y3,y2,0,0) be the binary representation of address Y, then B_A is computed as follows:
if ((y6,y5)==B_Z) then B_A=(y6,y5$\oplus$1) else B_A= (y6,y5)

This computation guarantees the prediction for a fetch block will be read from a different bank than that of the previous fetch block. The only information bits needed to compute the bank numbers for the two next fetch blocks A and B are bits *(y6,y5)*, *(z6,z5)* and B_Z: that is two-block ahead [18] bank number computation. These information bits are available one cycle before the effective access on the branch predictor is performed and the required computations are very simple. Therefore, no delay is introduced on the branch predictor by the bank number computation. In fact, bank selection can be performed at the end of Phase 1 of the cycle preceding the read of the branch predictor tables.

7 Indexing the branch predictor

As previously mentioned, the Alpha EV8 branch predictor is 4-way interleaved and the prediction and hysteresis tables are separate. Since the logical organization of the predictor contains the four *2Bc-gskew* components, this should translate to an implementation with 32 memory tables. However, the Alpha EV8 branch predictor only implements eight memory arrays: for each of the four banks there is an array for prediction and an array for hysteresis. Each word line in the arrays is made up of the four *logical* predictor components.

This section, presents the physical implementation of the branch predictor arrays and the constraints they impose on the composition of the indexing functions. The section also includes detailed definition of the hashing functions that were selected for indexing the different logical components in the Alpha EV8 branch predictor.

7.1 Physical implementation and constraints

Each of the four banks in the Alpha EV8 branch predictor is implemented as two physical memory arrays: the prediction memory array and the hysteresis memory array. Each word line in the arrays is made up of the four *logical* predictor components.

Each bank features 64 word lines. Each word line contains 32 8-bit prediction words from **G0**, **G1** and **Meta**, and 8 8-bit prediction words from **BIM**. A single 8-bit prediction word is selected from the word line from each predictor table **G0**, **G1**, **Meta** and **BIM**. A prediction read spans over 3 half cycle phases (5 phases including bank number computation and bank selection). This is illustrated in Fig. 3 and 4. A detailed description is given below.

1.**Wordline selection:** one of the 64 wordlines of the accessed bank is selected. The four predictor components share the 6 address bits needed for wordline selection. Furthermore, these 6 address bits can not be hashed since the wordline decode and array access constitute a critical path for reading the branch prediction array and consequently - inputs to decoder must be available at the very beginning of the cycle.

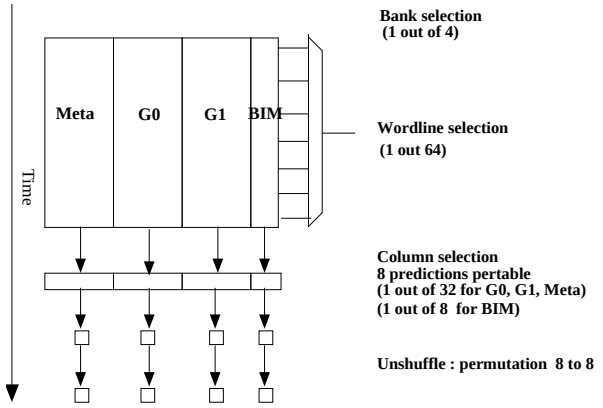

Figure 4. Reading the branch prediction tables

2. **Column selection:** each wordline consists of multiple 8-bit prediction entries of the four logical predictor tables. One 8-bit prediction word is selected for each of the logical predictor tables. As only one cycle phase is available to compute the index in the column, only a single 2-entry XOR gate is allowed to compute each of these column bits.

3. **Unshuffle:** 8-bit prediction words are systematically read. This word is further rearranged through a XOR permutation (that is bit at position i is moved at position $i \oplus f$). This final permutation ensures a larger dispersion of the predictions over the array (only entries corresponding to a branch instruction are finally useful). It allows also to discriminate between longer history for the same branch PC, since the computation of the parameter f for the XOR permutation can span over a complete cycle: each bit of f can be computed by a large tree of XOR gates.

Notations The three fetch-blocks old *lghist* history will be noted H= (h20, .., h0). A= (a52,..,a2,0,0) is the address of the fetch block. Z and Y are the two previous fetch blocks. I = (i15,..,i0) is the index function of a table, (i1,i0) being the bank number, (i4,i3,i2) being the offset in the word, (i10,i9,i8,i7,i6,i5) being the line number, and the highest order bits being the column number.

7.2 General philosophy for the design of indexing functions

When defining the indexing functions, we tried to apply two general principles while respecting the hardware implementation constraints. First, we tried to limit aliasing as much as possible on each individual table by picking individual indexing function that would spread the accesses over the predictor table as uniformly as possible. For each individual function, this normally can be obtained by mixing a large number of bits from the history and from the address to compute each individual bit in the function. However, general constraints for computing the indexing functions only allowed such complex computations for the unshuffle bits. For the other indexing bits, we favored the use of *lghist* bits instead of the address bits. Due to the inclusion of path information in *lghist*, *lghist* vectors were more uniformly distributed than PC addresses. In [17], it was pointed out that the indexing functions in a skewed cache should be chosen to minimize the number of block pairs that will conflict on two or more ways. The same applies for the *2Bc-gskew* branch predictor.

7.3 Shared bits

The indexing functions for the four prediction tables share a total of 8 bits, the bank number (2 bits) and the wordline number (i10, .., i5). The bank number computation was described in Section 6.

The wordline number must be immediately available at the very beginning of the branch predictor access. Therefore, it can either be derived from information already available earlier, such as the bank number, or directly extracted from information available at the end of the previous cycle such as the three fetch blocks old *lghist* and the fetch block address.

The fetch block address is the most natural choice, since it allows the use of an effective bimodal table for component **BIM** in the predictor. However, simulations showed that the distribution of the accesses over the **BIM** table entries were unbalanced. Some regions in the predictor tables were used infrequently and others were congested.

Using a mix of *lghist* history bits and fetch block address bits leads to a more uniform use of the different word lines in the predictor, thus allowing overall better predictor performance. As a consequence, component **BIM** in the branch predictor uses 4 bits of history in its indexing function.

The wordline number used is given by (i10,i9,i8,i7,i6,i5)= (h3,h2,h1,h0,a8,a7).

7.4 Indexing BIM

The indexing function for **BIM** is already using 4 history bits, that are three fetched blocks old, and some path information from two fetched block ahead (for bank number computation). Therefore path information from the last instruction fetch block (that is Z) is used. The extra bits for indexing **BIM** are (i13,i12,i11,i4,i3,i2)= (a11,$a9 \oplus a5$,$a10 \oplus a6$,a4,$a3 \oplus z6$,$a2 \oplus z5$).

7.5 Engineering the indexing functions for G0, G1 and Meta

The following methodology was used to define the indexing functions for **G0**, **G1** and **Meta**. First, the best history length combination was determined using standard skewing functions from [17]. Then, the column indices and the XOR functions for the three predictors were manually defined applying the following principles as best as we could: 1.favor a uniform distribution of column numbers for the choice of wordline index.

Column index bits must be computed using only one two-entry XOR gate. Since history vectors are more uniformly distributed than address numbers, to favor an overall good distribution of column numbers, history bits were generally preferred to address bits.
2. if, for the same instruction fetch block address A, two histories differ by only one or two bits then the two occurrences should not map onto the same predictor entry in any table : to guarantee this, whenever an information bit is X-ORed with another information bit for computing a column bit, at least one of them will appear alone for the computation of one bit of the unshuffle parameter.
3. if a conflict occurs in a table, then try to avoid it on the two other tables: to approximate this, different pairs of history bits are XORed for computing the column bits for the three tables.

This methodology lead to the design of the indexing functions defined below:

Indexing G0 *To simplify the implementation of column selectors,* **G0** *and* **Meta** *share i15 and i14.* Column selection is given by
(i15, i14, i13, i12, i11)= ($h7 \oplus h11$,$h8 \oplus h12$,$h4 \oplus h5$,$a9 \oplus h9$,$h10 \oplus h6$).
Unshufling is defined by (i4,i3,i2)= ($a4 \oplus a9 \oplus a13 \oplus a12 \oplus h5 \oplus h11 \oplus h8 \oplus z5$,$a3$⊕a11⊕h9 ⊕h10⊕h12⊕z6 ⊕a5, a2⊕a14⊕a10⊕h6 ⊕h4 ⊕h7 ⊕a6).

Indexing G1 Column selection is given by
(i15, i14, i13, i12, i11)= (h19⊕h12,h18⊕h11,h17⊕h10,h16⊕h4,h15⊕h20).
Unshuffling is defined by (i4,i3,i2)=
(a4⊕a11⊕a14⊕a6⊕h4⊕h6 ⊕h9⊕h14⊕h15⊕h16⊕z6, a3⊕a10⊕a13⊕h5⊕h11⊕h13⊕h18⊕h19⊕ h20⊕z5,a2 ⊕a5 ⊕a9⊕ h4 ⊕h8⊕h7⊕h10⊕h12⊕h13⊕h14⊕h17)

Indexing Meta Column selection is given by
(i15, i14, i13, i12, i11)= (h7⊕h11,h8⊕h12, h5⊕h13,h4⊕h9,a9⊕h6).
Unshuffling is defined by (i4,i3,i2)= (a4⊕a10⊕a5 ⊕h7⊕h10⊕h14⊕h13⊕z5, a3⊕a12⊕a14⊕a6⊕h4⊕ h6 ⊕h8 ⊕h14, a2⊕a9⊕ a11⊕a13⊕ h5⊕h9⊕h11⊕h12⊕z6)

8 Evaluation

In this section, we evaluate the different design decisions that were made in the Alpha EV8 predictor design. We first justify the choice of the hybrid skewed predictor *2Bc-gskew* against other schemes relying on global history. Then step by step, we analyze benefits or detriments brought by design decisions and implementation constraints.

8.1 Methodology

8.1.1 Simulation

Trace driven branch simulations with immediate update were used to explore the design space for the Alpha EV8 branch predictor, since this methodology is about three orders of magnitude faster than the complete Alpha EV8 processor simulation. We checked that for branch predictors using (very) long global history as those considered in this study, the relative error in number of branch mispredictions between a trace driven simulation, assuming immediate update, and the complete simulation of the Alpha EV8, assuming predictor update at commit time, is insignificant.

The metric used to report the results is mispredictions per 1000 instructions (misp/KI). To experiment with history length wider than log_2 of table sizes, indexing functions from the family presented in [17, 15] were used for all predictors, except in Section 8.5. The initial state of all entries in the prediction tables was set to weakly not taken.

8.1.2 Benchmark set

Displayed simulation results were obtained using traces collected with Atom[23]. The benchmark suite was SPECINT95. Binaries were highly optimized for the Alpha 21264 using profile information from the *train* input. The traces were recorded using the *ref* inputs. One hundred million instructions were traced after skipping 400 million instructions except for *compress* (2 billion instructions were skipped). Table 2 details the characteristics of the benchmark traces.

8.2 2Bc-gskew vs other global history based predictors

We first validated the choice of the *2BC-gskew* prediction scheme against other global prediction schemes. Fig. 5 shows simulation results for predictors with memorization size in the same range as the Alpha EV8 predictor. Displayed results assume *conventional branch history*. For all the predictors, the best history length results are presented. Fig. 6 shows the number of additional mispredictions for the same configurations as in Fig. 5 but using log_2 of the table size, instead of the best history length.

The illustrated configurations are:

- a 4*32K entries (i.e. 256 Kbits) *2Bc-gskew* using history lengths 0, 13, 16 and 23 respectively for **BIM**, **G0**, **Meta** and **G1**, and a 4*64K entries (i.e. 512Kbits) *2Bc-gskew* using history lengths 0, 17, 20 and 27. For limited(log_2) history length, the lengths are equal for all tables and are 15 for the 256Kbit configuration and 16 for the 512Kbit.
- a bimode predictor [13] consisting of two 128K entries tables for respectively biased taken and not taken branches and a 16 Kentries bimodal table, for a total of 544 Kbits of memorization[1]. The optimum history

[1]The original proposition for the bimode predictor assumes equal sizes for the three tables. For large size predictors, using a smaller bimodal table is more cost-effective. On our benchmark set, using more than 16K entries in the bimodal table did not add any benefit.

Benchmark	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
dyn. cond. branches (x1000)	12044	16035	11285	8894	16254	9706	13263	12757
static cond. branches	46	12086	3710	904	251	409	273	2239

Table 2. Benchmark characteristics

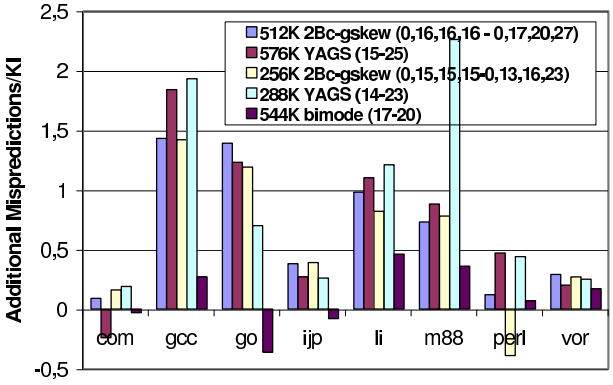

Figure 5. Branch prediction accuracy for various global history schemes

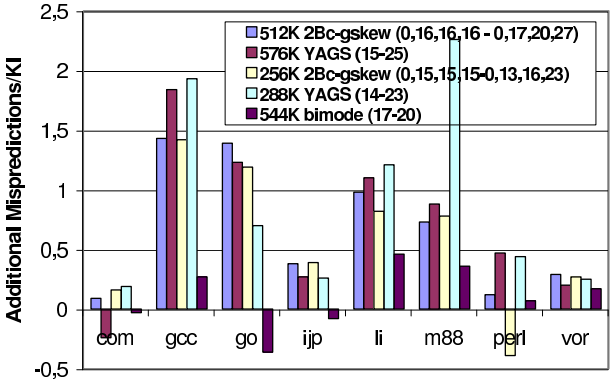

Figure 6. Additional Mispredictions when using log_2 table size history

length (for our benchmark set) was 20. For log_2 history length 17 bits were used.

- a 1M entries (2M bits) *gshare*. The optimum history length (on our benchmark set) was 20 (i.e log_2 of the predictor table size).
- a 288 Kbits and 576 Kbits YAGS predictor [4] (respective best history length 23 and 25) the small configuration consists of a 16K entry bimodal and two 16K partially tagged tables called direction caches, tags are 6 bits wide. When the bimodal table predicts taken (resp. not-taken), the *not-taken* (resp. *taken*) direction cache is searched. On a miss in the searched direction cache, the bimodal table provides the prediction. On a hit, the direction cache provides the prediction. For log_2 history length 14 bits (resp 15 bits) were used. 1

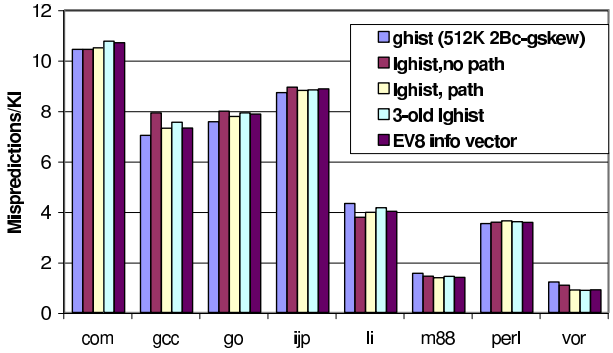

Figure 7. Impact of the information vector on branch prediction accuracy

First, our simulation results confirm that, at equivalent memorization budget *2Bc-gskew* outperforms the other global history branch predictors except *YAGS*. There is no clear winner between the *YAGS* predictor and *2Bc-gskew*. However, the *YAGS* predictor uses (partially) tagged arrays. Reading and checking 16 of these tags in only one and half cycle would have been difficult to implement. Second, the data support that, predictors featuring a large number of entries need very long history length and log_2 table size history is suboptimal.

8.3 Quality of the information vector

The discussion below examines the impact of successive modifications of the information vector on branch prediction accuracy assuming a 4*64K entries *2Bc-gskew* predictor. For each configuration the accuracies for the best history lengths are reported in Fig. 7. *ghist* represents the conventional branch history. *lghist,no path* assumes that *lghist* does not include path information. *lghist+path* includes path information. *3-old lghist* is the same as before, but considering three fetch blocks old history. *EV8 info vector* represents the information vector used on Alpha EV8, that is three fetch blocks old *lghist* history including path information plus path information on the three last blocks.

lghist As expected the optimal *lghist* history length is shorter than the optimal real branch history: (15, 17, 23) instead of (17, 20, 27) respectively for tables **G0**, **Meta** and **G1**. Quite surprisingly (see Fig. 7), *lghist* has same performance as conventional branch history. Depending on the application, there is either a small loss or a small benefit in accuracy. Embedding path information in *lghist* is gener-

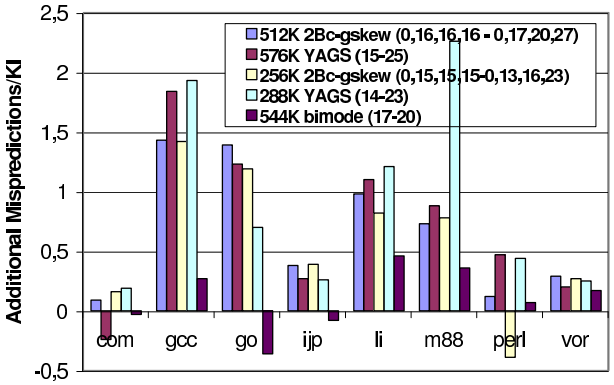

Figure 8. Adjusting table sizes in the predictor

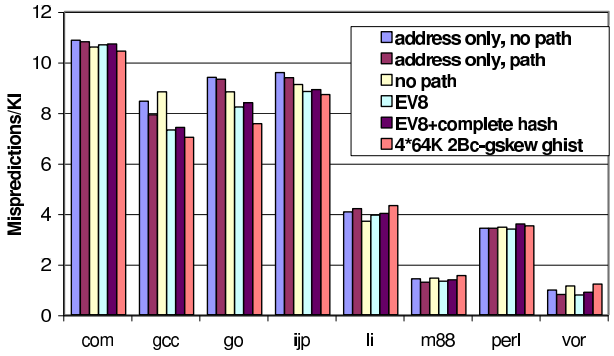

Figure 9. Effect of wordline indices

ally beneficial: we determined that is more often useful to de-alias otherwise aliased history paths.

The loss of information from branches in the same fetch block in *lghist* is balanced by the use of history from more branches (eventhough represented by a shorter information vector): for instance, for *vortex* the 23 *lghist* bits represent on average 36 branches. Table 3 represents the average number of conditional branches represented by one bit in *lghist* for the different benchmarks.

Three fetch blocks old history Using three fetch blocks old history slightly degrades the accuracy of the predictor, but the impact is limited. Moreover, using path information from the three fetch blocks missing in the history consistently recovers most of this loss.

EV8 information vector In summary, despite the fact that the vector of information used for indexing the Alpha EV8 branch predictor was largely dictated by implementation constraints, on our benchmark set this vector of information achieves approximately the same levels of accuracy as without any constraints.

8.4 Reducing some table sizes

Fig. 8 shows the effect of reducing table sizes. The base configuration is a 4*64K entries *2Bc-gskew* predictor (512Kbits). The data denoted by *small BIM* shows the performance when the **BIM** size is reduced from 64K to 16K 2-bit counters. The performance with a small BIM and half the size for **GO** and **Meta** hysteresis tables is denoted by *EV8 Size*. The latter fits the 352Kbits budget of the Alpha EV8 predictor. The information vector used for indexing the predictor is the information vector used on Alpha EV8.

Reducing the size of the **BIM** table has no impact at all on our benchmark set. Except for *go*, the effect of using half size hysteresis tables for **G0** and **Meta** is barely noticeable. *go* presents a very large footprint and consequently is the most sensitive to size reduction.

8.5 Indexing function constraints

Simulations results presented so far did not take into account hardware constraints on the indexing functions. 8 bits of index must be shared and can not be hashed, and computation of the column bits can only use one 2-entry XOR gate Intuitively, these constraints should lead to some loss of efficiency, since it restricts the possible choices for indexing functions.

However, it was remarked in [16] that (for caches) partial skewing is almost as efficient as complete skewing. The same applies for branch predictors: sharing 8 bits in the indices does not hurt the prediction accuracy as long as the shared index is uniformly distributed.

The constraint of using unhashed bits for the wordline number turned out to be more critical, since it restricted the distribution of the shared index. Ideally for the EV8 branch predictor, one would desire to get the distribution of this shared 8 bit index as uniform as possible to spread accesses on **G0**, **G1** and **Meta** over the entire table.

Fig. 9 illustrates the effects of the various choices made for selecting the wordline number. *address only, no path* assumes that only PC address bits are used in the shared index **and** that no path information is used in *lghist*. *address only, path* assumes that only PC address bits are used in the shared index, but path information is embedded in *lghist*. *no path* assumes 4 history bits and 2 PC bits as wordline number, but that no path information is used in *lghist*. *EV8* illustrates the accuracy of the Alpha EV8 branch predictor where 4 history bits are used in the wordline number index **and** path information is embedded in the history. Finally *complete hash* recalls the results assuming hashing on all the information bits and *4*64K 2Bc-gskew ghist* represents the simulation results assuming a 512Kbits predictor with no constraint on index functions and conventional branch history.

Previously was noted that incorporating path information in *lghist* has only a small impact on a *2Bc-gskew* predictor indexed using hashing functions with no hardware constraints. However, adding the path information in the history for the Alpha EV8 predictor makes the distribution of *lghist* more uniform, allows its use in the shared index

	compress	gcc	go	ijpeg	li	m88ksim	perl	vortex
lghist/ghist	1.24	1.57	1.12	1.20	1.55	1.53	1.32	1.59

Table 3. Ratio lghist/ghist

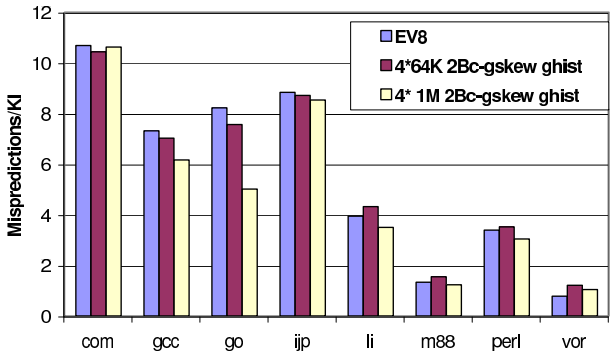

Figure 10. Limits of using global history

and therefore can increase prediction accuracy.

The constraint on the column bits computation indirectly achieved a positive impact by forcing us to very carefully design the column indexing and the unshuffle functions. The (nearly) total freedom for computing the unshuffle was fully exploited: 11 bits are XORed in the unshuffling function on table *G1*. The indexing functions used in the final design outperform the standard hashing functions considered in the rest of the paper: these functions (originally defined for skewed associative caches [17])exhibit good interbank dispersion, but were not manually tuned to enforce the three criteria described in Section 7.5.

To summarize, the 352 Kbits Alpha EV8 branch predictor stands the comparison against a 512 Kbits *2Bc-gskew* predictor using conventional branch history.

9 Conclusion

The branch predictor on Alpha EV8 was defined at the beginning of 1999. It features 352 Kbits of memory and delivers up to 16 branch predictions per cycle for two dynamically succesive instruction fetch blocks. Therefore, a global history prediction scheme had to be used. In 1999, the hybrid skewed branch predictor *2Bc-gskew* prediction scheme [19] represented state-of-the-art for global history prediction schemes. The Alpha EV8 branch predictor implements a *2Bc-gskew* predictor scheme enhanced with an optimized update policy and the use of different history lengths on the different tables.

Some degrees of freedom in the definition of *2Bc-gskew* were tuned to adapt the predictor parameters to silicon area and chip layout constraints: the bimodal component is smaller than the other components and the hysteresis tables of two of the other components are only half-size of the predictor tables.

Implementation constraints imposed a three fetch blocks old compressed form of branch history, *lghist*, instead of the effective branch history. However, the information vector used to index the Alpha EV8 branch predictor stands the comparison with complete branch history. It achieves that by combining path information with the branch outcome to build *lghist* and using path information from the three fetch blocks that have to be ignored in *lghist*.

The Alpha EV8 is four-way interleaved and each bank is single ported. On each cycle, the branch predictor supports requests from two dynamically succesive instruction fetch blocks but does not require any hardware conflict resolution, since bank number computation guarantees **by construction** that any two dynamically succesive fetch blocks will access two distinct predictors banks.

The Alpha EV8 branch predictor features four logical components, but is implemented as only two memory arrays, the prediction array and the hysteresis array. Therefore, the definition of index functions for the four (logical) predictor tables is strongly constrained: 8 bits must be shared among the four indices. Furthermore, timing constraints restrict the complexity of hashing that can be applied for indices computation. However, efficient index functions turning around these constraints were designed.

Despite implementation and size constraints, the Alpha EV8 branch predictor delivers accuracy equivalent to a 4*64K entries *2Bc-gskew* predictor using conventional branch history for which no constraint on the indexing functions was imposed.

In future generation microprocessors, branch prediction accuracy will remain a major issue. Even larger predictors than the predictor implemented in the Alpha EV8 may be considered. However, this brute force approach would have limited return except for applications with a very large number of branches. This is exemplified on our benchmark set in Fig. 10 that shows simulation results for a 4*1M 2-bit entries *2Bc-gskew* predictor. Adding back-up predictor components [3] relying on different information vector types (local history, value prediction [9, 6], or new prediction concepts (e.g., perceptron [11]) to tackle hard-to-predict branches seems more promising. Since such a predictor will face timing constraints issues, one may consider further extending the hierarchy of predictors with increased accuracies and delays: line predictor, global history branch prediction, backup branch predictor. The backup branch predictor would deliver its prediction later than the global history branch predictor.

References

[1] B. Calder and D. Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[2] K. Diefendorff. Compaq Chooses SMT for Alpha. *Microprocessor Report*, December 1999.

[3] K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceeding of the 30th Symposium on Microarchitecture*, Dec. 1998.

[4] A. N. Eden and T. Mudge. The YAGS branch predictor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, Dec 1998.

[5] G. Giacalone and J. Edmonson. Method and apparatus for predicting multiple conditional branches. In *US Patent No 6,272,624*, August 2001.

[6] J. Gonzalez and A. Gonzalez. Control-flow speculation through value prediction for superscalar processors. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[7] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, October 1996.

[8] E. Hao, P.-Y. Chang, and Y. N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, California, 1994.

[9] T. Heil, Z. Smith, and J. E. Smith. Improving branch predictors by correlating on data values. In *32nd Int. Symp. on Microarchitecture*, Nov. 1999.

[10] S. Hily and A. Seznec. Branch prediction and simultaneous multithreading. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, Oct. 1996.

[11] D. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, January 2001.

[12] T. Juan, S. Sanjeevan, and J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 155–166, New York, June 27–July 1 1998. ACM Press.

[13] C.-C. Lee, I.-C. Chen, and T. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Dec 1997.

[14] S. McFarling. Combining branch predictors. Technical report, DEC, 1993.

[15] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, June 1997.

[16] A. Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[17] A. Seznec and F. Bodin. Skewed associative caches. In *Proceedings of PARLE' 93*, May 1993.

[18] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 116–127, 1996.

[19] A. Seznec and P. Michaud. De-aliased hybrid branch predictors. Technical Report RR-3618, Inria, Feb 1999.

[20] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-Level Parallelism, vol. 2, Jan. 2000*, January 2000.

[21] J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[22] E. Sprangle, R. S. Chappell, M. Alsup, and Y. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 284–291, June 1997.

[23] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 29(6):196–205, 1994.

[24] A. Talcott, M. Nemirovsky, and R. Wood. The influence of branch prediction table interference on branch prediction scheme performance. In *Proceedings of the 3rd Annual International Conference on Parallel Architectures and Compilation Techniques*, 1995.

[25] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, June 1995.

[26] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice : Instruction fetch and issue on an implementable simultaneous MultiThreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, May 1996.

[27] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[28] C. Young, N. Gloy, and M. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

Acknowledgement

The authors would like to recognise the work of all those who contributed to the architecture, circuit implementation and verification of the EV8 Branch Predictor. They include Ta-chung Chang, George Chrysos, John Edmondson, Joel Emer, Tryggve Fossum, Glenn Giacalone, Balakrishnan Iyer, Manickavelu Balasubramanian, Harish Patil, George Tien and James Vash.