Dynamic branch prediction and control speculation

Article *in* International Journal of High Performance Systems Architecture · January 2007

DOI: 10.1504/IJHPSA.2007.013287

CITATIONS
9

READS
3,045

3 authors, including:

Jurij Silc
Jožef Stefan Institute
121 PUBLICATIONS **1,409** CITATIONS

SEE PROFILE

Borut Robic
University of Ljubljana
133 PUBLICATIONS **1,349** CITATIONS

SEE PROFILE

Dynamic branch prediction and control speculation

Jurij Šilc*

Department of Computer System,
Jožef Stefan Institute, Ljubljana, Slovenia
E-mail: jurij.silc@ijs.si
*Corresponding author

Theo Ungerer

Institute of Computer Science,
University of Augsburg, Germany
E-mail: ungerer@informatik.uni-augsburg.de

Borut Robič

Faculty of Computer and Information Science,
University of Ljubljana, Slovenia
E-mail: borut.robic@fri.uni-lj.si

Abstract: Branch prediction schemes have become an integral part of today's superscalar processors. They are one of the key issues in enhancing the performance of processors. Pipeline stalls due to conditional branches are one of the most significant impediments to realise the performance potential of superscalar processors. Many schemes for branch prediction, that can effectively and accurately predict the outcome of branch instructions have been proposed. In this paper, an overview of some dynamic branch prediction schemes for superscalar processors are presented.

Keywords: superscalar processor; branch prediction; high performance; simulation; prototype.

Reference to this paper should be made as follows: Šilc, J., Ungerer, T. and Robič, B. (2007) 'Dynamic branch prediction and control speculation', *Int. J. High Performance Systems Architecture*, Vol. 1, No. 1, pp.2–13.

Biographical notes: Jurij Šilc is the Deputy Head of the Department of Computer Systems at the Jožef Stefan Institute, Ljubljana, Slovenia and an Associative Professor at the Jožef Stefan Postgraduate School, Ljubljana, Slovenia. His research interests include processor architecture, parallel computing, combinatorial and numerical optimisation.

Theo Ungerer is the Chair of Systems and Networking at the University of Augsburg, Germany. Since 2003, he is also Scientific Director of the Computing Center of the University of Augsburg. His current research interests are in the areas of embedded processor architectures, embedded real-time systems, ubiquitous systems and organic computing.

Borut Robič is Professor at the Faculty of Computer and Information Science, University of Ljubljana, Slovenia. His main research interests are in algorithms, computational complexity and parallel computing.

1 Introduction

Excellent branch handling techniques are essential for current and future microprocessors. Many instructions are in different stages in the pipeline of a wide-issue superscalar processor. Instruction issue also works best with a large instruction window, leading to even more instructions that are 'inflight' in the pipeline. However, approximately every seventh instruction in an instruction stream is a branch instruction which potentially interrupts the instruction flow through the pipeline.

The task of high performance branch handling consists of the following requirements:

1 an early determination of the branch outcome (the so-called branch resolution)

2 buffering of the branch target address in a branch target address cache after its first calculation and an immediate reload of the program counter after a branch target address cache match

3 an excellent branch predictor (i.e. branch prediction technique) and speculative execution mechanism

4 often another branch is predicted while a previous branch is still unresolved, so the processor must be able to pursue two or more speculation levels and

5 an efficient rerolling mechanism when a branch is mispredicted (minimising the branch misprediction penalty).

An early branch resolution is supported by forwarding as soon as possible to the branch instruction the results of compare instructions that may be stored in a general-purpose register or in a special condition-code register. Branch testing could be moved forward in the pipeline as far as the instruction decode stage. Previous calculations of branch target addresses are cached in a branch target address cache and accessed during the instruction fetch stage.

The performance of branch prediction depends on the prediction accuracy and the cost of misprediction. Prediction accuracy can be improved by inventing better branch predictors, but the misprediction penalty depends on many organisational features: the pipeline length (favouring shorter over longer pipelines), the overall organisation of the pipeline, whether misspeculated instructions can be removed from internal buffers or have to be executed and can only be removed in the retire stage.

2 Static branch prediction

Static branch prediction is a simple prediction technique which either always uses a fixed prediction direction or allows the compiler to determine the prediction direction. The prediction direction of a branch instruction is never changed. Simple hardware-fixed direction mechanisms can be.

Predict always not taken: this is the simplest scheme because the assumption is a straight instruction flow. Unfortunately, due to frequent loops in an instruction flow, this technique is not very effective. This prediction technique should not be confused with the delayed branch technique (a popular technique in the first generations of scalar RISC processors). The instruction in the delay slot is always executed, while the predict-not-taken-technique executes the instructions after the branch speculatively and squashes the instruction execution in the case of misprediction.

Predict always taken: here branches at the end of a loop iteration are correctly predicted as long as the loop loops. The branch target address has to be stored within the instruction fetch unit to allow a zero delay.

Backward branch predict taken, forward branch predict not taken: here the idea is that branches with branch target addresses pointing backwards stem from loops and should be predicted taken, while other kind of branches are preferably not taken.

Sometimes a bit in the branch opcode allows the compiler to decide the prediction direction either directly (bit set means 'predict taken', bit not set means 'predict not taken') or by reversing the hardware-determined direction.

The compiler may use several techniques for a good compiler-based static prediction. It may either:

1 examine the program structure for prediction (branches at the end loop iteration code should be predicted as taken, if-then branches predicted as not taken)

2 relegate prediction to the programer by compiler directives or

3 use a profile-based prediction by predicting the branch directions based on prior runs of the program with recording of the branch behaviour.

The profile-based prediction is nearly always better than the simpler direction-based predictions.

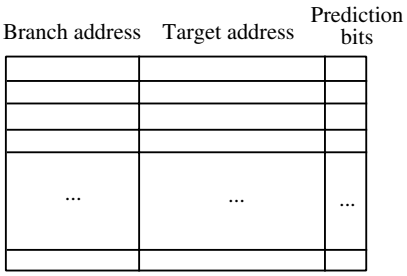
3 Dynamic branch prediction

In a dynamic branch prediction scheme, prediction is decided on the computation history of the program. After a start-up phase of the program execution, where a static branch prediction might be effective, historical information is gathered and dynamic branch prediction becomes more effective. In general, dynamic gives better results than static branch prediction, but at the cost of increased hardware complexity.

3.1 Branch-target buffer

The branch target address is needed at the same time as the prediction. In particular, it should be known already in the IF stage whether the as-yet-undecoded instruction is a (conditional or unconditional) branch to allow an instruction fetch at the target address in the next cycle. The Branch-Target Buffer (BTB) is a branch-prediction cache that stores the predicted address for the next instruction after a branch (Lee and Smith, 1984). The BTB is accessed during the IF stage. It consists of a table with branch addresses, the corresponding target addresses, and prediction information (see Figure 1 for a simple BTB). The PC for the next instruction to fetch is compared with the entries in the BTB. If a matching entry is found in the BTB, fetching can start immediately at the target address.

Figure 1 Branch-target buffer



The BTB stores branch and jump target addresses. Branch target addresses are predicted addresses, while jump target addresses always transfer control. Jumps (unconditional branches) are usually much less frequent than conditional branches.

Fetching instructions from a new target address is fast if the fetch address hits in the I-cache.[1] Moreover, for procedure calls and returns a small stack of return addresses is often used in addition to and independent of, a BTB. Such a *return*

address stack appears, for example, in the Alpha 21164 organised as a 12-entry circular buffer that makes the last 12 return addresses available.

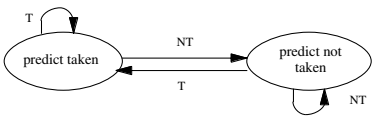
3.2 One-bit predictor

The simplest dynamic branch prediction scheme is a simple Branch History Table (BHT). The BHT is a small buffer memory containing branch addresses indexed by the lower bits of the address of a branch instruction. Each entry of the BHT contains one bit that indicates whether the branch was recently taken or not. If the bit is set, the branch is predicted taken. If the bit is not set, the branch is predicted not taken. In the case of a misprediction, the bit state is reversed and so is the prediction direction.

One-bit predictors can also be implemented in the BTB by only storing the target addresses of predicted taken branches.

The prediction states of a 1-bit predictor are shown in Figure 2 (T stands for taken and NT stands for not taken).

Figure 2 One-bit predictor states



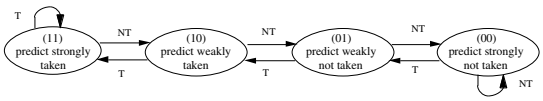
Such a 1-bit predictor correctly predicts a branch at the end of a loop iteration, as long as the loop does not exit. However, in nested loops, a 1-bit prediction scheme will cause two mispredictions for the inner loop: one at the end of the loop, when the iteration exits the loop instead of looping again and one when executing the first loop iteration, when it predicts exit instead of looping. Such a double misprediction in nested loops is avoided by a 2-bit predictor scheme.

3.2.1 Two-bit predictors

In a 2-bit prediction scheme 2 bits instead of one are assigned to each entry in the BHT. The 2 bits stand for the prediction states 'predict strongly taken', and 'predict weakly taken', 'predict strongly not taken', 'predict weakly not taken'. In the case of a misprediction in the 'strongly' state cases, the prediction direction is not changed, rather the prediction goes into the respective 'weakly' state. A prediction must miss twice before it is changed when a 2-bit prediction scheme is applied.

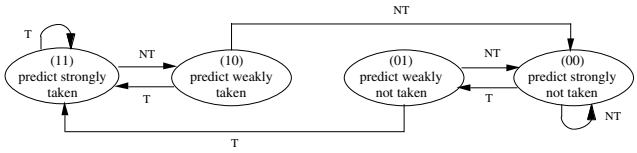
Two kinds of 2-bit prediction schemes are used: the saturation up-down counter scheme demonstrated in Figure 3 and the scheme given in Figure 4.

Figure 3 Two-bit predictor saturation counter states



In the 2-bit saturation up-down counter scheme, the counter is incremented for each taken branch occurrence and decremented each time the branch is not taken. The counter is saturating, that is, it is not decremented past 0, nor is it incremented past 3. The most significant bit determines the prediction.

Figure 4 Two-bit predictor states



The other scheme given in Figure 4 differs from the saturation up-down counter scheme by changing directly from the 'weakly' to the 'strongly' states in the case of a second misprediction. This scheme is applied in the UltraSPARC-I processor. Branches without prediction are initialised by the UltraSPARC-I processor to 'predict weakly not taken' (Tremblay and O'Connor, 1996).

Hennessy and Patterson (1996) showed that the mispredictions of SPEC89 programs vary from 1% (`nasa7`, `tomcat`) to 18% (`eqntott`), with `spice` at 9% and `gcc` at 12%, assuming a 4096-entry BHT.

The 2-bit prediction scheme is extendable to a n-bit scheme. However, studies have shown that a 2-bit prediction scheme does almost as well as a n-bit scheme with $n > 2$.

Two-bit predictors can be implemented in the BTB, assigning two state bits to each entry in the BTB. Another solution, which is proposed for the PowerPC 604 and 620, is to use a BTB for target addresses and a BHT as a separate prediction buffer. While the BTB is accessed in the IF stage, the BHT prediction is performed in the PowerPC 604 and 620 one cycle later in the ID stage and may override the previous BTB prediction.

A mispredict in the BHT occurs for two reasons: either a wrong guess for that branch or the branch history of a wrong branch is used because of the way the table is indexed. In an indexed table lookup, part of the instruction address is used as an 'index' to identify a table entry. Instruction addresses with the same bit pattern used as an index share the same table entry, leading to frequent mispredicts if the table is small.

Two-bit predictors work well for scientific floating-point intensive programes which contain many frequently executed loop-control branches. Shortcomings of the 2-bit prediction schemes arise from dependent (correlated) branches, which are frequent in integer-dominated programes.

The following example of two branches, one dependent on the other, demonstrates that 1-bit and 2- predictors can potentially mispredict every time. Let us look at the following program (Hennessy and Patterson, 1996):

```
if (d == 0)    /* branch b1 */
    d = 1;
if (d == 1)    /* branch b2 */
    ...
```

In assembly language notation the program can be given as follows (variable d is assigned to register $R1$):

```
     bnez R1,L1     ; branch b1 (d ≠ 0)
     addi R1,R0,#1  ; d == 0, so d = 1
L1:  subi R3,R1,#1
     bnez R3,L2     ; branch b2 (d ≠ 0)
     ...
L2:  ...
```

Consider a sequence where d alternates between 0 and 2 which generates a sequence of NT-T-NT-T-NT-T for branches b1 and b2. The execution behaviour is given in the following table:

Initial d	$d \stackrel{?}{==} 0$	*b1*	*d before b2*	$d \stackrel{?}{==} 1$	*b2*
0	yes	NT	1	yes	NT
2	no	T	2	no	T

If we apply a 1-bit predictor which is initialised to 'predict taken' for branches b1 and b2, then every branch is mispredicted. The same behaviour is shown for the 2-bit predictor of Figure 3 starting from the state 'predict weakly taken'. The 2-bit predictor of Figure 4 mispredicts every second branch execution of b1 and b2. A (1, 1)-correlating predictor (see below) can take advantage of the correlation between the two branches; it mispredicts only in the first iteration when $d = 2$.

Correlating branch predictors usually reach higher prediction rates for integer-intensive programs than the 2-bit predictor scheme and require only a small increase in hardware cost.

3.3 Correlation-based predictors

The 2-bit predictor scheme only uses the recent behaviour of a single branch to predict the future of that branch. Correlations between different branch instructions are not taken into account. Let us also look at the recent behaviour of other branches rather than just the branch we are trying to predict.

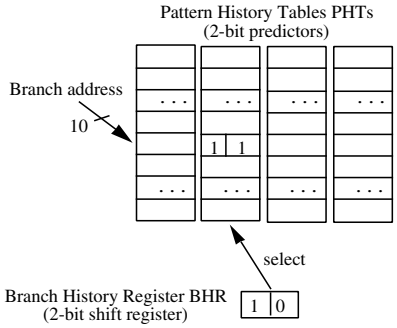
The so-called *correlation-based predictors* developed by Pan et al. (1992) are branch predictors that additionally use the behaviour of other branches to make a prediction. While 2-bit predictors use self-history only, the correlating predictor uses neighbour history as well. Many integer workloads feature complex control-flows whereby the outcome of a branch is affected by the outcomes of recently executed branches. In other words, the branches are correlated (Pan et al., 1992).

A correlation-based predictor denoted in short as an (m, n)*-predictor*, uses the behaviour of the last m branches to choose from 2^m branch predictors, each of which is a n-bit predictor for a single branch. The global history of the most recent m branches can be recorded in a m-bit shift register – called a Branch History Register (BHR) – where each bit records whether the branch was taken or not taken. Each time a branch in execution resolves, its sign bit is shifted into the BHR. The contents of the BHR are used to address (index) the entries in a so-called *Pattern History Table* (PHT).[2] Typically 2-bit predictors are used in PHTs.

A (1, 1)-predictor uses the behaviour of the last branch to choose between a pair of 1-bit predictors and a correlation-based predictor denoted as a (2, 2)-predictor uses a BHR of 2 bits to choose among four 2-bit prediction tables. A 2-bit predictor (without global history) can simply be denoted as a (0, 2)-predictor.

Figure 5 shows the implementation of correlation-based predictor, a type (2, 2)-predictor with four 1 k-entry PHTs. The BHR bit pattern selects the specific PHT. The entries of the 1 k-entry PHTs are generally accessed by using the lower order 10 bits of the branch address. Depending on the implementation, the PHTs may alternatively be accessed using 10 bits of the address of the instruction immediately prior to the branch under consideration (Pan et al., 1992). The four 1 k-entry PHTs can also be viewed as a single 4 k-entry PHT. Then 12 bits are required for the PHT lookup. Therefore, 2 bits from the BHR are concatenated with 10 bits from the branch address.

Figure 5 Implementation of a (2, 2)-predictor



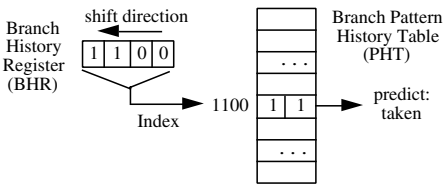
3.4 Two-level adaptive predictors

The two-level adaptive predictor was developed by Yeh and Patt (1992) at the same time as the closely related correlation-based prediction scheme. There are several variations of the two-level adaptive prediction scheme Yeh and Patt (1993).

The basic two-level predictor uses a single 'global' BHR of k bits to index in a PHT of 2-bit counters. The global BHR is updated with outcomes from all branches. Thus, not only the history of a branch, but also the history of other branches, influence the prediction of the branch. All schemes that use a single global BHR are called *global history schemes* and correspond to Pan et al.'s correlation-based predictor schemes.

In the simplest case there is a single global BHR (denoted G) and a single global PHT (denoted g), this simple predictor is called GAg (A stands for 'adaptive'). All PHT implementations of Yeh and Patt use 2-bit predictors. An implementation of a GAg-predictor with a 4-bit BHR length (therefore, also denoted as GAg(4)) is shown in Figure 6.[3] The BHR is implemented as a simple shift register shifting right to left with the sign (1 for branch taken, 0 for branch not taken) of the last resolved branch at the rightmost bit position.

Figure 6 Implementation of a GAg(4)-predictor



In the GAg predictor scheme the PHT lookup depends entirely on the bit pattern in the BHR and is completely independent of the branch address. The advantages of the 'degenerate' GAg scheme are its simple implementation and

the fact that the predicted outcome of a branch can be known long before the execution of that branch (Pan et al., 1992).

A simple GAg(k)-predictor often performs better on integer programs than a 2-bit-predictor (with a saturation up-down counter scheme).

However, GAg-predictors still suffer from branch patterns that emerge several times within a computation. Two code sequences may have the same bit pattern in the BHR and thus index the same pattern in the PHT. Since the branch behaviour of the two code sequences may differ, the shared pattern may lead to the wrong predictions.

Such wrong predictions can be restrained by additionally using:

1 the (full) branch address to distinguish multiple PHTs (called per-address PHTs)

2 a subset of branches (e.g. defined by part of the branch address) to distinguish multiple PHTs (called per-set PHTs)

3 the (full) branch address to distinguish multiple BHRs (called per-address BHRs)

4 a subset of branches to distinguish multiple BHRs (called per-set BHRs) or

5 a combination scheme.

In the first two cases, a single global BHR is combined with multiple per-address selected PHTs, denoted as GAp or with multiple per-set addressed PHTs, denoted as GAs. A GAp predictor with a 4-bit BHR, denoted as GAp(4), is shown in Figure 7 and a GAs predictor with a 4-bit BHR, denoted as GAs (4, 2^n), is shown in Figure 8. In the GAs (4, 2^n) predictor n bits of the branch address are used to define 2^n different branch sets corresponding to 2^n PHTs with 2^4 entries each. Branches of the same branch set share the same PHT in a GAs predictor.

Figure 7 Implementation of a GAp(4) predictor

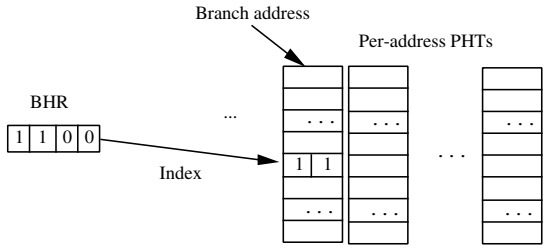
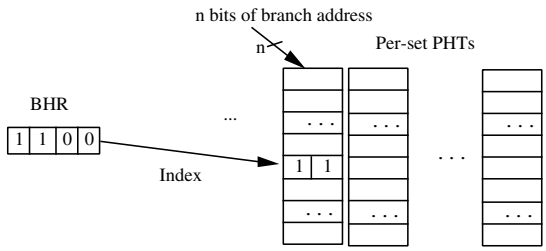


Figure 8 Implementation of a GAs(4, 2^n) predictor



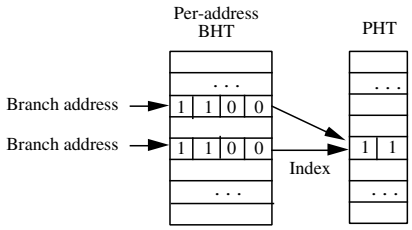
The three two-level adaptive predictors GAg, GAp and GAs use a single global BHR and together form the *global history scheme* predictors. These predictors are closely related to the correlation-based predictor.

In fact, by rotating Figure 5 90° to the right and assuming a 4-bit BHR, it can be seen that a correlation-based (4, 2)-predictor is equivalent to a GAs(4) predictor, assuming $n = 10$ bits in the branch address (compare with Figure 8).

A second scheme class is defined as the *per-address history schemes* where the first-level branch history refers to the last k occurrences of the same branch instruction (using self-history only !). Therefore, a BHR is associated with each branch instruction to distinguish the branch history information of each branch. The BHRs record self-history in contrast to the neighbour-history recording BHR used in global history schemes. The per-address BHRs are combined in a table which is called the Per-address Branch History Table (PBHT) by Yeh and Patt.

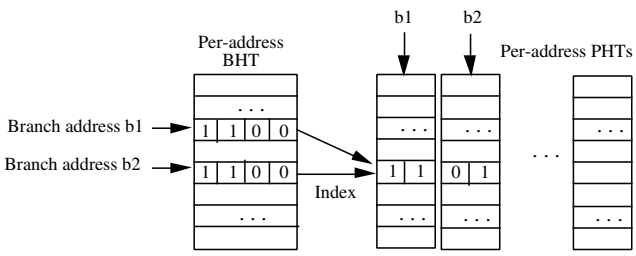
In the simplest per address history scheme, the BHRs index into a single global PHT. Such a two-level adaptive predictor is denoted as PAg (multiple per-address indexed BHRs and a single global PHT). An implementation of a PAg(4) predictor is shown in Figure 9. Two different branches with the same BHT bit pattern select the same PHT entry leading to unnecessary misprediction.

Figure 9 Implementation of a PAg(4) predictor



The combination of multiple per-address BHRs with multiple per-address PHTs, denoted as a PAp predictor and of multiple per-address BHRs with multiple per-set PHTs, denoted as a PAs predictor, is also possible. In the PAp scheme each branch has its own BHR and its own PHT. So the number of BHRs in the per-address BHT and the number of PHTs is equal. However, the numbers are not fixed. They depend on the number of branches in the programe.[4] Conceptually, the BHR content is used as an index to select an entry in its PHT. The PHT is selected by the branch instruction address (PAp) or by the branch set (PAs). An implementation of a PAp(4) predictor is shown in Figure 10. The figure shows the case of two branches with the same BHT bit pattern that indexes the same line in the per-address PHTs. However, the branch addresses select different PHTs and thus different PHT entries.

Figure 10 Implementation of a PAp(4) predictor



In the per-address history schemes only the execution history of the branch itself has an effect on its prediction. The branch

prediction is non-correlating – independent of the execution history of other branches.

In the *per-set history schemes* the first-level branch history means the last k occurrences of the branch instructions from the same subset. Each BHR is associated with a set of branches. The set attributes of a branch can be determined by the branch opcode, the branch class which is assigned by the compiler or by part of the branch address. Since a per-set addressed BHR is potentially updated with history from all branches in the same set, the prediction of a branch is influenced by other branches in the same set (Yeh and Patt, 1993). Again the three variations are determined by the variations in the organisation of the second-level, namely SAg, SAs and SAp. Implementations of a SAg(4) and a SAs(4) predictor are shown in Figures 11 and 12. Figure 11 shows that the SAg-predictor may suffer from branch patterns that emerge several times within a computation (the same bit pattern in the BHRs select the same PHT entry in the global PHT). Moreover, in all per-set history schemes, branches which fall into the same set (e.g. having the same n bits in the branch address) select the same entry in the BHT (and/or the same PHT). This is demonstrated in Figure 12.

Figure 11 Implementation of a SAg(4) predictor

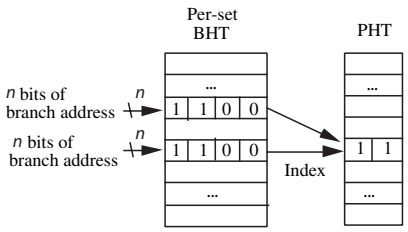
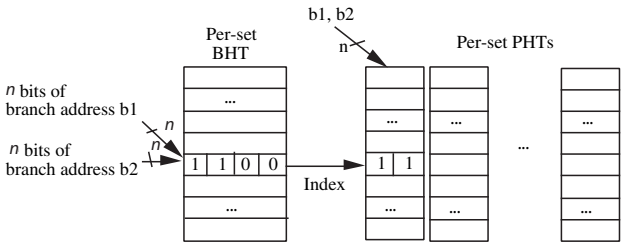


Figure 12 Implementation of a SAs(4) predictor



The full list of Yeh and Patt's two-level adaptive branch predictors is given as follows (Yeh and Patt, 1993):

- GAg: global BHR, global PHT
- GAs: global BHR, per-set PHTs
- GAp: global BHR, per-address PHTs
- PAg: per-address BHT, global PHT
- PAs: per-address BHT, per-set PHTs
- PAp: per-address BHT, per-address PHTs
- SAg: per-set BHT, global PHT
- SAs: per-set BHT, per-set PHTs and
- SAp: per-set BHT, per-address PHTs.

The denotation of the two-level adaptive branch predictors are derived from the following table which gives a simplified estimation of the hardware costs (Yeh and Patt, 1993):

Scheme name	*BHR length*	*No. of PHTs*	*Hardware cost*
GAg(k)	k	1	$k + 2^k \times 2$
GAs(k, p)	k	p	$k + p \times 2^k \times 2$
GAp(k)	k	b	$k + b \times 2^k \times 2$
PAg(k)	k	1	$b \times k + 2^k \times 2$
PAs(k, p)	k	p	$b \times k + p \times 2^k \times 2$
PAp(k)	k	b	$b \times k + b \times 2^k \times 2$
SAs(k)	k	1	$s \times k + 2^k \times 2$
SAs(k, $s \times p$)	k	p	$s \times k + p \times 2^k \times 2$
SAp(k)	k	b	$s \times k + b \times 2^k \times 2$

In the table b is the number of PHTs or entries in the BHT for the per-address schemes. p and s denote the number of PHTs or entries in the BHT for the per-set schemes, assuming that different per-set schemes are possible for BHR selection and for PHT selection.

The simulations of Yeh and Patt (1993) using the SPEC89 benchmarks show that the performance of the global history schemes is sensitive to the branch history length. Interference of different branches that are mapped to the same PHT is decreased by lengthening the global BHR leading to better prediction accuracy. Similarly adding PHTs reduces the possibility of pattern history interference by mapping interfering branches into different tables.

In general, the global history schemes are better than the per-address schemes for the integer SPEC89 programs, while the per-address schemes are better for the floating-point intensive programs. The phenomenon is due to the ability of the global history schemes to utilise branch correlation, which is often the case in the frequent if-then-else statements in integer programs, while the per-address schemes are better in predicting loop-control branches which are frequent in the floating-point SPEC89 benchmark programs. The per-set history schemes are in between other schemes.

Comparing the cost effectiveness of the different schemes using the formulas in the table given above and a fixed hardware budget of $8\,k$ bits, the most cost-effective global history scheme is GAs(7, 32), the best per-address scheme is PAs(6, 16) and for per-set schemes SAs(6, 4 $\times$16) scores best. From these three configurations PAs(6, 16) achieves the highest average prediction accuracy.

When given a higher hardware budget of $128\,k$ bits, the most cost-effective global history scheme is GAs(13, 32), the best per-address scheme is PAs(8, 256) and the best per-set scheme is SAs(9, 4 $\times$ 32). Of these configurations GAs(13, 32) achieves the highest measured prediction accuracy of 97.2 %.[5]

Yeh and Patt conclude that global history schemes perform better than other schemes on integer-dominated programs but require higher implementation costs to be effective overall. However, in the global history schemes, the pattern history of different branches interfere with each other if they map to the same PHT. Therefore, long BHRs and/or many PHTs should be used.

Per-address history schemes perform better than other schemes on floating-point programes. Per-set history schemes have a performance that is similar to global

history schemes on integer programs and similar to per-address schemes on floating-point intensive programs.

3.5 gselect and gshare predictors

McFarling (1993) analysed the 2-bit predictors and correlation-based predictor schemes and introduced a number of new predictors. One set of new correlation-based predictors uses a hash function into the PHT instead of indexing the PHT to reduce conflicts.

Recall that in the correlation-based predictor scheme the (2, 2)-predictor shown in Figure 5 requires 12 bits for a PHT table lookup (assuming a single unified PHT instead of the four PHTs); 2 bits from the BHR are concatenated with 10 bits from the branch address. McFarling calls this bit concatenation in a correlation-based or GAs predictor the *gselect* predictor which concatenates some lower order bits of the branch address and of the bit pattern in the BHR.

In contrast to simple indexing, McFarlings's *gshare* predictor uses the bitwise exclusive OR of part of the branch address and the BHR as a hash function. To demonstrate the ability of both predictor types, McFarling uses the following table:

Branch address	*BHR*	*gselect 4/4*	*gshare 8/8*
00000000	00000001	00000001	00000001
00000000	00000001	00000000	00000000
11111111	00000000	11110000	11111111
11111111	10000000	11110000	01111111

Strategy *gselect* 4/4 concatenates the lower order 4 bits of the branch address with the lower order 4 bits of the BHR. Strategy *gshare* 8/8 uses the bitwise XOR of all 8 bits of both the branch address and the BHR. Comparing *gshare* 8/8 and *gselect* 4/4 shows that only *gshare* is able to separate all four cases. The *gselect* predictor cannot take advantage of the distinguishing history in the upper 4 bits of the BHR.

3.6 Hybrid predictors

The second strategy proposed by McFarling (1993) is to combine multiple separate branch predictors, each tuned to a different class of branches. Different branch prediction schemes have different advantages. Hopefully, such a *combining predictor* achieves an even better prediction accuracy than either of the predictors used for combination. To predict a given branch, typically two or more predictors and a predictor selection mechanism are necessary in a combining predictor. In principle, all kinds of branch predictors are candidates for combination of predictors.

McFarling combined the 2-bit predictor[6] with the *gshare* two-level adaptive predictor and concluded that, in this combination, global information can be used if it is worthwhile; otherwise, the usual branch direction as predicted by the 2-bit predictor can be used. Another combination proposed by the same author is the combination of a PAp predictor[7] with the *gshare* scheme. Simulations with SPEC89 benchmarks showed that both hybrid predictors outperform the *gshare* which itself is better than *gselect* and all other predictors for a given counter array size.

Another kind of *hybrid predictor* proposed by Young and Smith (1994) combines a compiler-based static branch prediction with a dynamic predictor of the two-level adaptive type. Profiling is used to collect the static prediction information (Uht et al., 1997).

Grunwald et al. (1998) compared the SAg, *gshare* and McFarling's combining predictor (combining a two-bit predictor with the *gshare* predictor) using the SPECint95 benchmarks. The results are reported in the Table 1. The table shows that for SPECint95 benchmark programes about every sixth instruction of the trace (the executed and committed instructions) is a branch instruction and in the mean misprediction rate the combining predictor performs best with 8.1% mispredictions. Further simulation showed that the processor typically issued 20–100% more instructions than actually commit, due to speculative execution (Grunwald et al., 1998).

Other simulations by Keeton et al. (1998) using an OLTP (online transaction workload) on a Pentium Pro multiprocessor reported a misprediction ratio of 14% with a branch instruction frequency of about 21%. The speculative execution factor, given by the number of instructions decoded divided by the number of instructions committed, is 1.4 for the database programs.

Two different conclusions may be drawn from these simulation results: branch predictors should be improved further and/or branch prediction is only effective if the branch is predictable. If a branch outcome is dependent on irregular data inputs, as is often the case in OLTP applications or game-playing programs, the branch often shows an irregular behaviour. This may be the reason for the high misprediction rate of the SPECint95 benchmark program `go`.

Numerous other selector and hybrid predictor types are evaluated and reported in the research literature.

3.7 Confidence estimation

If a branch is not or is not easily, predictable, its irregular behaviour will frequently yield costly misspeculations. The predictability of branches can be assessed by additionally measuring the confidence in the prediction. A *low confidence branch* is a branch which frequently changes its branch direction in an irregular way making its outcome hard to predict or even unpredictable.

Confidence estimation is a technique for assessing the quality of a particular prediction. If applied to branch prediction, a confidence estimator attempts to assess the prediction made by a branch predictor. Because each branch is eventually determined to have been predicted correctly or incorrectly, the confidence estimator assigns a 'high confidence' or a 'low confidence' to each prediction. In combination with the two prediction outcomes 'correctly predicted' and 'incorrectly predicted', four confidence classes can be measured:

1 correctly predicted with high confidence

2 correctly predicted with low confidence

3 incorrectly predicted with high confidence and

4 incorrectly predicted with low confidence.

Table 1 SAg, *gshare* and McFarling's combining predictor

Application	*Committed instructions (in millions)*	*Conditional branches (in millions)*	*Taken branches (%)*	*Misspredict rate (%)*		
				SAg	*gshare*	*Combining*
compress	80.4	14.4	54.6	10.1	10.1	9.9
gcc	250.9	50.4	49.0	12.8	23.9	12.2
perl	228.2	43.8	52.6	9.2	25.9	11.4
go	548.1	80.3	54.5	25.6	34.4	24.1
m88ksim	416.5	89.8	71.7	4.7	8.6	4.7
xlisp	183.3	41.8	39.5	10.3	10.2	6.8
vortex	180.9	29.1	50.1	2.0	8.3	1.7
jpeg	252.0	20.0	70.0	10.3	12.5	10.4
mean	267.6	46.2	54.3	8.6	14.5	8.1

When a branch is actually resolved, the branch can be classified as belonging to one of these classes (Grunwald et al., 1998).

To implement a confidence estimator, information from the branch prediction tables is used. Already in early 1980s, Smith (1981) proposed to use saturation counter information to construct a confidence estimator. The concept was to speculate more aggressively when the confidence level is higher (Smith, 1998). Jacobsen et al. (1996) used a *miss distance counter table* in addition to the branch predictor. Each time a branch is predicted, the value in the table is compared to a threshold. If the value is above the threshold, then the branch is considered to have high confidence and low confidence otherwise. Tyson et al. (1997) observed that a small number of branch history patterns typically leads to correct predictions in a PAs predictor scheme. Their confidence estimator assigned high confidence to a fixed set of patterns and low confidence to all others (Grunwald et al., 1998).

Confidence estimation can be used for speculation control provided that ways other than branch speculation can be used to utilise the processor resources. Such alternative ways can be, for example, thread switching in multithreaded processors (Ungerer et al., 2002, 2003) or multipath execution where instructions from both branch directions are fetched and executed, and the wrong path instructions are afterwards discarded. In a simultaneous multithreaded processor (Ungerer et al., 2002, 2003), it may be more cost effective to switch threads than speculatively evaluate a branch of low confidence. In a multipath execution model both branch paths of a low confidence branch may be evaluated, whereas a conventional branch speculation may be employed to high confidence branches. Both techniques need the ability of a processor to pursue two different instruction streams simultaneously. Because of the limitation of a single instruction pointer in today's superscalar processors, such techniques are confined to multithreaded processors and related processor techniques such as multiscalar and trace processors (Šilc et al., 2000).

3.8 Predicated instructions

One technique that allows us to 'evaluate' two branch paths in a multiple-issue processor is *predication* (August et al ., 1997, 1998; Hwu, 1998; Mahlke et al., 1995). Using this technique, the instruction set architecture of a processor is enhanced by so-called *predicated* or *conditional instructions* and one or more *predicate registers*. The Boolean result of a condition test is recorded in a (1-bit) predicate register. Predicated instructions use a predicate register as an additional input operand.

Predication is demonstrated by the following source code sequence:

if $(x == 0)$ { /* branch b1 */
$a = b + c;$
$d = e - f;$ }
$g = h * i;$ /* instruction independent of branch b1 */

Translation of the example source code sequence, using a branch instruction for the alternative, would lead to a speculative execution with instruction $g = h * i$ and all later instructions on the speculative path of branch b1. In the case of a misspeculation temporary results of this and all later instructions would be unnecessarily discarded. However, the source code is translated in the following code sequence using predicated instructions (each line represents a single machine operation):

$(Pred = (x == 0))$
if $Pred$ **then** $a = b + c;$
if $Pred$ **then** $d = e - f;$
$g = h * i;$

As can be seen from the example, predication is able to eliminate a branch and therefore, the associated branch prediction, increasing the distance between mispredictions. Also the run length of a code block is increased which allows better instruction scheduling by an optimising compiler. However, the compiler must assure that the exception behaviour is not changed by moving the instruction across a set-predicate instruction.

Predication affects the instruction set, adds a port to the register file and complicates instruction execution. Predication is most effective when control dependencies can be completely eliminated, such as in an if-then with a small then body and when the condition can be evaluated early. The use of predicated instructions is limited when the control flow involves more than a simple alternative sequence. Moreover, predicated instructions that are discarded still consume processor resources; the fetch bandwidth is especially affected.

If the full instruction set is predicated (a so-called full predication model), predication bits in the opcode are

additionally needed for each instruction to denote a predicate register. Thus, often only a few instructions of the instruction set architecture, in most cases the load instructions, are predicated instructions.

Most signal processors, high-performance microcontrollers and some contemporary superscalar processors employ predication. As examples, the ARM processor and the Intel Merced are fully predicated, while Alpha, MIPS, PowerPC and SPARC processors use conditional move instructions.

Predicated instructions are fetched, decoded and placed in the instruction window like non-predicated instructions. It depends on the processor architecture how far a predicated instruction proceeds speculatively in the pipeline before its predication is resolved:

- A predicated instruction executes only if its predicate is true, otherwise the instruction is discarded. In this case predicated instructions are not executed before the predicate is resolved.
- Alternatively, as reported for Intel's IA-64 instruction set architecture, the predicated instruction may be executed, but commits only if the predicate is true, otherwise the result is discarded (Dulong, 1998).

The latter case is similar to the eager or multipath execution model described below.

3.9 Eager execution

With the *eager* or *multipath* execution model, execution proceeds down both paths of a branch and no prediction is made. When a branch resolves, all operations on the non-taken path are discarded. Consequently, eager execution with unlimited resources, which can be characterised as 'oracle execution', would give the same theoretical maximum performance as a perfect branch prediction. With limited resources, the eager execution strategy must be employed carefully. Resource consumption rises exponentially with each level of branches that are executed eagerly. Therefore, instead of employing full eager execution, a mechanism is required that decides when to employ prediction and when eager execution.

One decision mechanism is the use of a confidence estimator. If a branch prediction can be made with high confidence, branch prediction and single path speculative execution is employed; when low confidence is the case, eager execution spares the misprediction penalty.

Until now, the eager execution strategy has rarely been implemented, except for limited applications, such as instruction fetch in the SuperSPARC processor and in the IBM 360/91 (Uht et al., 1997) and subsequent IBM mainframes, for example, the IBM 3090 processor.

The 'nanothreaded' DanSoft processor implements a multipath-execution model using confidence information from a static branch prediction mechanism (Gwennap, 1997).

A number of research projects have surveyed eager execution. The PolyPath architecture (Klauser et al., 1998) enhances a superscalar processor by a limited multipath execution feature to employ eager execution. Heil and Smith (1996) propose selective dual path execution and Tyson et al. (1997) propose a limited dual path execution. Wallace et al. (1998) survey threaded multipath execution, employing eager execution in a simultaneous multithreaded processor model.

Unger et al. (1998) propose a compiler technique called *simultaneous speculation scheduling* in combination with a 'minimal' multithreaded execution model to enable speculative execution of alternative program paths. The technique is only applicable for architectures that fulfil certain requirements of a base multithreaded processor model:

- First, the processor must be able to pursue two or more threads of control concurrently, that is, it must provide two or more independent program counters.
- All concurrently executed threads of control share the same address space, preferably the same register set.
- The instruction set must provide a number of thread-handling instructions: here the minimal requirements for multithreading are an instruction for creating a new thread (fork) and an instruction that conditionally stops its own execution or the execution of some other threads (sync).
- Creating a new thread by the `fork` instruction and joining threads by the sync instruction must be extremely fast, preferably single-cycle operations.

Uht and Sindagi (1995) propose the *disjoint eager execution* technique. The idea is to assign resources to branch paths whose results are most likely to be used, that is, branches with the highest cumulative execution probability. Uht and Sindagi's notion of branch execution probability is closely related to the confidence in a branch prediction, for which they use the branch prediction accuracy, that is, the percentage of taken or untaken executions of a branch.

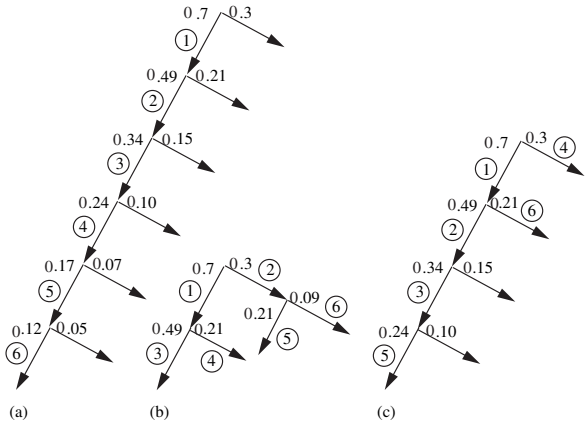
While a branch path is speculatively executed, further branches may be encountered before the first branch resolves, often resulting in a branch speculation level of four or more. The cumulative execution probability accumulates the prediction accuracies of a branch and of the pending (predicted but yet to be resolved) branches of previous speculation levels. If all branches in such a sequence of pending branches are simply assumed to be independent of each other, the single prediction accuracies can be multiplied to determine the cumulative execution probability of the last branch in the sequence.

Thus in the disjoint eager execution model, all branches are predicted, the cumulative prediction accuracy is computed and compared to the accuracies of all branch paths that were yet to be chosen for speculative execution. The branch path with the highest cumulative prediction accuracy is executed, leading to either another single path speculative execution or an eager execution.

The three different possibilities of single path speculative execution as produced by the usual speculation methods described above, full eager execution and disjoint eager execution are demonstrated in Figure 13 (Uht et al., 1997). Each line with an arrow represents a branch path marked by its cumulative probability. For illustration, branch prediction accuracy is 70% for each individual branch. All branches are

pending. Branch paths with circled numbers are in execution, branch paths that are not chosen by the prediction are the paths without circled numbers. Circled numbers indicate the order of the resource assignment, that is, the order in which the paths are speculatively assigned. Figure 13(c) shows that the disjoint eager execution strategy allocates resources to more likely branch paths than the single path and the eager execution models.

Figure 13 (a) Single path speculative execution, (b) full eager execution and (c) disjoint eager execution



3.10 Prediction of indirect branches

All branch prediction techniques reported above are directed towards prediction of direct branches, whose targets are encoded in the instruction itself. Indirect branches, which transfer control to an address stored in a register, are even harder to predict accurately. Though indirect branches are not as frequent as direct branches in C− or FORTRAN-benchmark programs, indirect branches occur with higher frequency in machine code compiled from object-oriented programs like C++ and Java. Virtual function tables, used in C++ and Java compilers to implement late binding of subroutine invocations, execute an indirect branch for every polymorphic call. A simple BTB is a poor predictor for branches with changing targets. One simple possibility is to update the PHT to include the branch target addresses.

Driesen and Hoelzle (1998) reported an indirect branch frequency of once every 50 instructions for several large object-oriented C++ programs. They investigated two-level and hybrid indirect branch predictors and reported a misprediction rate of 10% with a 1 *k*-entry table, 7% with an 8 *k*-entry table, 9% for a 1 *k*-entry hybrid predictor and 6% in the 8 *k*-entry hybrid predictor case.

3.11 High-bandwidth branch prediction

Future microprocessors will require more than one prediction per cycle starting speculation over multiple branches in a single cycle. Here the GAg scheme is able to predict multiple branches without knowing the branch instruction address. However, the instruction fetch is also affected. When multiple branches are predicted per cycle, instructions must be fetched from multiple target addresses per cycle, complicating I-cache access. A trace cache (Rotenberg et al., 1996) in combination with next trace prediction is able to solve both problems by fetching from a dynamically assembled trace line, rather than from I-cache.

A combination of branch handling techniques will most likely be applied, such as a multihybrid branch predictor (Evers et al., 1996; Patt et al., 1997) combined with support for context switching, indirect jumps and interference handling. As already emphasised, a fast and accurate branch prediction is essential for advanced superscalar processors with hundreds of in-flight instructions. Branch prediction itself is already a well-developed part of microarchitecture design. One observation is that many branches display different characteristics that cannot be optimally predicted by a single-scheme branch predictor. Evers et al. (1996) propose hybrid branch predictors, a technique that was previously proposed by combining two predictors (McFarling, 1993) and that is already implemented in the PowerPC 620. Hybrid predictors comprise several predictors, each targeting different classes of branches. The principal idea is that each predictor scheme works best for a particular branch type.

As predictor tables increase in size, they often take more time to react to changes in a program (warm-up time). A hybrid predictor with several components can solve this problem by using component predictors with shorter warm-up times while the larger predictors are warming up. Examples of predictors with shorter warm-up times are two-level predictors with shorter histories as well as smaller dynamic predictors (Patt et al., 1997).

The *multihybrid branch predictor* (Evers et al., 1996; Patt et al., 1997) uses a set of selection counters for each entry in the branch target buffer, in the trace cache or in a similar structure, keeping track of the predictor currently most accurate for each branch and then using the prediction from that predictor for that branch. The multihybrid predictor performs better than regular hybrid predictors. It reaches a prediction rate of 95% for 16 kB predictor size and up to near 97% for 256 kB predictors using programs of the SPECint95 benchmark suite (Patt et al., 1997). Despite this high prediction rate, the remaining mispredictions still incur a large performance penalty. Other branch techniques must be combined with branch prediction. Such techniques are predication to enlarge the number of instructions between two speculative predictions or both-path execution (as in the PolyPath architecture by Klauser et al. (1998)) in the case of low branch prediction confidence (Grunwald et al., 1998).

3.12 Neural branch predictors

The first dynamic neural branch predictors were proposed by Vintan and Iridon (1999). The neural branch predictor research was consistently developed further by Jiménez and Lin (2001, 2002) with the first *perceptron predictor*, feasible to be implemented in hardware. The main advantage of the neural predictor consists in its ability to exploit long histories requiring linear resources growth, while classical predictors are requiring exponential resources growth. The main disadvantage of the perceptron predictor consists in its high latency.

In order to reduce the prediction latency, a *fast-path neural predictor* was proposed (Jiménez, 2003). Here, a perceptron predictor choosing its weights for generating a prediction

according to the current branch's path, rather than according to the branch's program counter.

Many studies have extended the perceptron predictor. Loh and Henry (2002) use the perceptron predictor as a component of a larger hybrid predictor. Thomas et al. (2003) find salient history bits for the perceptron predictor using dynamic data-flow analysis. Akkary et al. (2004) adapt the perceptron predictor to provide confidence estimates for speculation control. Falcón et al. (2004) use a perceptron predictor as a component of a prophet/critic hybrid predictor that runs the branch predictor ahead to second-guess previous predictions and possibly reverse them.

The neural branch predictor concept is very promising and Intel already implements this idea in one of the Itanium's simulators for researching future microarchitectures.

4 Conclusions

When a branch is not predicted correctly, there is rarely a penalty of less than two cycles, even in simple RISC pipelines. However, the misprediction penalty depends on many organisational features: the pipeline length (favouring shorter over longer pipelines), the overall organisation of the pipeline, whether misspeculated instructions can be removed from internal buffers or have to be executed and can only be removed in the retire stage. Further dynamic aspects that influence the misprediction penalty are the number of speculative instructions in the instruction window or the reorder buffer. Typically only a limited number of instructions can be removed each cycle. Therefore, rerolling when a branch is mispredicted is typically expensive, for example, 11 or more cycles in the Pentium II or the Alpha 21264 processors. The high misprediction penalty in current and prospective future microprocessors shows the importance of excellent branch prediction mechanisms for the overall performance of a processor.

Beside static and dynamic branch prediction they are other techniques to handle branches. For example, predication using so-called predicated or conditional instructions that allow the removal of the branch from the instruction flow and eager execution of both branch sides. Eager execution is especially effective when the branch direction changes in an irregular fashion which means the branch is not predictable. In that case the expensive rerolling mechanism slows down execution. However, eager execution is not possible with today's superscalar processors because the ability to pursue two instruction streams in parallel is necessary.

References

Akkary, H., Srinivasan, S.T., Koltur, R., Patil, Y. and Refaai, W. (2004) 'Perceptron-based branch confidence estimation for speculation control', *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, Madrid, Spain, pp.265–275.

August, D.I., Hwu, W.W. and Mahlke, S.A. (1997) 'A framework for balancing control flow and predication', *Proceedings of the 30th Annual International Symposium on Microarchitecture*, Research Triangle Park, NC, pp.92–103.

August, D.I., et al. (1998) 'Integrated Predicated and speculative execution in the IMPACT EPIC architecture', *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, pp.227–237.

Driesen, K. and Hoelzle, U. (1998) 'Accurate indirect branch prediction', *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, pp.167–177.

Dulong, C. (1998) 'The IA-64 architecture at work', *Computer*, Vol. 3, pp.24–31.

Evers, M., Chang, P-Y. and Patt, Y.N. (1996) 'Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches', *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, PA, pp.3–11.

Falcón, A., Stark, J., Ramirez, A., Lai, K. and Valero, M. (2004) 'Prophet/critic hybrid branch prediction', *Proceedings of the 31st Annual International Symposium on Computer Architecture*, München, Germany, pp.250–263.

Grunwald, D., Klauser, A., Manne, S. and Pleszkun, A. (1998) 'Confidence estimation for speculation control', *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, pp.122–131.

Gwennap, L. (1997) 'DanSoft develops VLIW design', *Microprocessor Report*, Vol. 11, No. 2, pp.18–22.

Heil, T.H. and Smith, J.E. (1996) 'Selective dual path execution', *Technical Report*, ECE, University of Wisconsin-Madison.

Hennessy, J.L. and Patterson, D.A. (1996) *Computer Architecture a Quantitative Approach*, 2nd edition, San Mateo, CA: Morgan Kaufmann.

Hwu, W.W. (1998) 'Introduction to predicated execution', *Computer*, Vol. 31, pp.49–50.

Jacobsen, E., Rotenberg, E. and Smith, J.E. (1996) 'Assigning confidence to conditional branch predictions', *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Paris, France, pp.142–152.

Jiménez, D.A. (2003) 'Fast path-based neural branch predictor', *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, San Diego, CA, pp.243–252.

Jiménez, D.A. and Lin, C. (2001) 'Dynamic branch prediction with perceptrons', *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, Monterrey, Mexico, pp.197–206.

Jiménez, D.A. and Lin, C. (2002) 'Neural methods for dynamic branch prediction', *ACM Transactions on Computer Systems*, Vol. 20, No. 4, pp.369–397.

Keeton, K., Patterson, D.A., He, Y.Q., Raphael, R.C. and Baker, W.E. (1998) 'Performance characterization of a quad pentium Pro SMP using OLTP workloads', *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, pp.15–26.

Klauser, A., Paithankar, A. and Grunwald, D. (1998) 'Selective eager execution on the PolyPath architecture', *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, pp.250–259.

Lee, J.K.F. and Smith, A.J. (1984) 'Branch prediction strategies and branch target buffer design', *Computer*, Vol. 17, pp.6–22.

Loh, G.H. and Henry, D.S. (2002) 'Predicting conditional branches with fusion-based hybrid predictors', *Proceedings of the 11th Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, pp.165–176.

Mahlke, S.A., Hank, R.E., McCormick, J.E., August, D.I. and Hwu, W-M.W. (1995) 'A comparison of full and partial predicated execution support for ILP processors', *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, pp.138–149.

McFarling, S. (1993) 'Combining branch predictors', *WRL Technical Notes TN-36*, Digital Western Research Laboratory.

Pan, S-T., So, K. and Rahmeh, J.T. (1992) 'Improving the accuracy of dynamic branch prediction using branch correlation', *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, pp.76–84.

Patt, Y.N., Patel, S.J., Evers, M., Friendly, D.H. and Stark, J. (1997) 'One billion transistors, one uniprocessor, one chip', *Computer*, Vol. 30, pp.51–57.

Rotenberg, E., Bennett, S. and Smith, J.E. (1996) 'Trace cache: a low latency approach to high bandwidth instruction fetching', *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Paris, France, pp.24–34.

Šilc, J., Ungerer, T. and Robič, B. (2000) 'A survey of new research directions in microprocessors', *Microprocessors and Microsystems*, Vol. 24, No. 4, pp.175–190.

Smith, J.E. (1981) 'A study of branch prediction strategies', *Proceedings of the Eighth Annual Symposium on Computer Architecture*, Minneapolis, MI, pp.135–147.

Smith, J.E. (1998) 'Retrospective: a study of branch prediction techniques', *25 Years of the International Symposia on Computer Architecture. Selected Papers*, ACM Press, pp.22–23.

Thomas, R., Franklin, M., Wilkerson, C. and Stark, J. (2003) 'Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history', *Proceedings of the 30th Annual International Symposium on Computer Architecture*, San Diego, CA, pp.314–323.

Tremblay, M. and O'Connor, J.M. (1996) 'UltraSPARC I: a four-issue processor supporting multimedia', *IEEE Micro*, Vol. 16, pp.42–50.

Tyson, G., Lick, K. and Farrens, M. (1997) 'Limited dual path execution', *Technical Report CSE-TR 346-97*, University of Michigan.

Uht, A.K. and Sindagi, V. (1995) 'Disjoint eager execution: an optimal form of speculative execution', *Proceedings of the 28th International Symposium on Microarchitecture*, Ann Arbor, MI, pp.313–325.

Uht, A.K., Sindagi, V. and Somanathan, S. (1997) 'Branch effect reduction techniques', *Computer*, Vol. 30, pp.71–81.

Unger, A., Ungerer, T. and Zehendner, E. (1998) 'Static speculation, dynamic resolution', *Proceedings of the Seventh Workshop on Compilers for Parallel Computers*, Linkoping, Sweden, pp.243–253.

Ungerer, T., Robič, B. and Šilc, J. (2002) 'Multithreaded processors', *The Computer Journal*, Vol. 45, No. 3, pp.320–348.

Ungerer, T., Robič, B. and Šilc, J. (2003) 'A survey of processors with explicit multithreading', *ACM Computing Surveys*, Vol. 35, No. 1, pp.29–63.

Vintan , L.N. and Iridon, M. (1999) 'Towards a high performance neural branch predictor', *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, Washington, DC, pp.868–873.

Wallace, S., Calder, B. and Tullsen, D.M. (1998) 'Threaded multiple path execution', *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Spain, pp.238–249.

Yeh, T-Y. and Patt, Y.N. (1992) 'Alternative implementation of two-level adaptive branch prediction', *Proceedings of the 19th Annual Symposium on Computer Architecture*, Gold Coast, Australia, pp.124–134.

Yeh, T-Y. and Patt, Y.N. (1993) 'A comparison of dynamic branch predictors that use two levels of branch history', *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, CA, pp.257–266.

Young, C. and Smith, M. (1994) 'Improving the accuracy of static branch prediction using branch correlation', *Proceedings of the Sixth Annual International Symposium on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, pp.232–241.

Notes

[1]A variation of the BTB that was popular for older processors without on-chip I-caches is to store one or more target instructions additionally to the target address.

[2]Pan et al. (1992) used the terms 'branch prediction table' instead of 'PHT' and '*m*-bit shift register' instead of 'BHR'.

[3]The GAg scheme is called the 'degenerate case' of the correlation scheme by Pan et al. (1992).

[4]The PAp predictor is mainly of theoretical interest, because the variable numbers of BHRs and PHTs cause implementation problems.

[5]Prediction accuracy measured for SPECint95 or OLTP (online transaction processing) programes is much lower than for SPEC89 benchmarks (see Table 1).

[6]Called a bimodal predictor by McFarling (1993).

[7]Called a local predictor by McFarling, per-address scheme in Yeh and Patt's nomenclature.