N. A. Ismail, M. Abdullah and F. A. Torkey, "Performance study of dynamic branch prediction schemes for modern ILP processors," El-Azhar University Engineering Journal, vol. 5, No....

Article · January 2002

CITATIONS
0

READS
594

2 authors, including:

Nabil A. Ismail
Faculty of Electronic Engineering Menoufia University
209 PUBLICATIONS **624** CITATIONS

SEE PROFILE

Performance Study of Dynamic Branch Prediction Schemes for Modern ILP Processors

Nabil A. Ismail[*] *Senior Member, IEEE,* **Mohamed A. Abdullah**[*], and **Fawzy A. Torkey**[**]

***Abstract*—Modern processors use superscalar architectures with deep-pipelines in order to execute multiple instructions per cycle. Performance of these instruction-level parallel (ILP) processors is governed by the smooth flow of valid instructions in the pipelines. Increase in the number of the pipeline stages increases the probability of encountering branch instructions during execution. The frequency and behavior of branch instructions seriously hinder performance of modern ILP processors. History of the branches coupled with direction patterns is captured by branch predictors to predict the direction before the instruction is resolved. Various mechanisms, both at the compiler as well as the processor level, have been proposed to predict the branch behavior. In this work, we have investigated various dynamic branch predictors at processor level to do a fair comparison among them. We have used a common set of Spec95 benchmarks and. A practical implementation is described using eight SPECint95 benchmarks and similar key parameters for evaluating these predictors. Some of the largest performance gains are observed on *go* and *gcc* benchmarks, which have traditionally posed the most difficult challenge to aggressive branch prediction techniques**

***Index Terms*—branch misprediction penalty, superscalar, branch predictor, speculative execution, processor performance.**

I. Introduction

MICROARCHITECTURE research has demonstrated the effectiveness of multiple hardware contexts for improving throughput, hiding memory latency and supporting thread-level (TLP) and instruction-level parallelism (ILP) in state-of-the-art processor implementations. The ILP paradigm has been exploited using combinations of several high performance techniques: deep pipelines [1], VLIW execution [1], superscalar out-of-order [1,2,3], decoupling [4], multithreading [5], the trace-based proposals [6,7], and the instruction level distributed processing (ILDP) discussed in [8].

The current generation of microprocessors all use deep pipelines and superscalar execution [1,2,3] coupled with a complex memory hierarchy based on several cache levels to attempt executing more instructions per cycle (> 2.0 IPC). At the same time, achieve higher clock frequencies. This in turn motivates microarchitectures with wider machine widths, deeper pipelines and higher complexity. The original Pentium processor, for example, had a relatively short pipeline of five stages. This grew to 12 stages in the highly successful Pentium Pro/II/III series of processors, and the recent Pentium 4 uses 20 stages.

General-purpose code has many conditional branches, irregular control flow, and much less data parallelism. These code characteristics and their detrimental consequences, in the form of branch effects, have severely limited the parallelism that can be exploited. The ability to minimize stalls or pipeline bubbles that may result from branches is becoming increasingly critical as microprocessor designs implement grater degrees of ILP.

There are three techniques for dealing with the conditional branch problem. The first, and most widely studied, is the speculative execution technique which aims to improve branch prediction. This approach has received considerable (successful) research effort for many years. The second is to fetch and execute both paths following a branch, and keep only the computation of the correct path. Of course, this can lead to exponential growth in hardware, so recently, more selective approaches have been advocated, where multi-path execution is only used for hard-to-predict branches [9,10]. Predicated execution (also called conditional execution or guarded execution) [11,12,13], is a software method for achieving a similar effect. The third approach is aimed at reducing the penalty after a misprediction occurs. This approach exploits the fact that not all instructions following a mispredicted branch have performed useless computation [14].

In speculative execution, code is executed before the result of the branch is known. In order to decide which path to speculatively execute, the processor must "predict" whether the branch will be taken or not taken. If the processor has predicted incorrectly, all speculative work beyond a branch must be discarded. Although speculative execution most refers to *simple-path*, because they execute down one path from a branch, *eager execution* and the most recent *disjoint eager execution* are also possible [9]. As the pipeline depths and the issue rates increase, the amount of speculative work that must be thrown away on the event of a branch misprediction also increases.

When a branch misprediction is detected in a traditional superscalar processor, the processor performs a series of steps to ensure correct execution. Instructions after the mispredicted branch are squashed and all resources they hold are freed. Typically, freeing resources includes returning physical registers to the freelist and reclaiming entries in the instruction issue buffers, reorder buffer, and load/store queues. In addition, the mapping of physical registers is backed up to the point of the mispredicted branch. The instruction fetch unit is also backed up to the point of the mispredicted branch and the processor begins sequencing on the correct path.

[*]Nabil. A. Ismail and Mohamed A. Abdullah are with the Computer Science & Engineering Department, Faculty of Electronic Engineering, Menoufyia University, Menouf, 32953, Egypt, (e-mail: nabil_is@hotmail.com).

[**]Fawzy A. Torkey is the Dean of the Faculty of Computers and Information, Menoufyia University, Shebin-El-Kum, Egypt.

Therefore, a very accurate *branch predictor* is essential for achieving high performance on deeply pipelined superscalar processors. Hence the greater the distance between mispredictions, the more parallelism can be extracted. For example, improving branch prediction accuracy from just 85% to 90% increases the distance between mispredictions by 50%, as given by:

$$\text{Distance} \propto 1/(1 - \text{accuracy})$$

Not only branch directions but also target addresses must be predicted. The branch target buffer (BTB) which is a form of cache is commonly used to handle the branch's target address through hardware.

The performance loss due to branch instructions was first approached with ***static branch predictors***, which always predict the same outcome for a given branch [15]. This prediction was obtained either using very simple heuristics [16], static analysis [17], or profile information [18]. The profile-guided branch prediction achieves the highest prediction accuracy among the static predictors, correctly predicting about 87% of the branches.

As the transistor budget in the modern processor increased, branch prediction moved to the more accurate ***dynamic predictors***. Dynamic branch prediction algorithms use information gathered at run-time to predict branch directions. Various dynamic branch prediction schemes have been studied in the past decade [19,20,21], with each offering certain distinctive features. Most of them, however, share a common characteristic: they rely on a collection of 1- or 2-bit counters held in a *predictor table* known as *pattern history table* (PHT). Each entry in the table records the recent outcomes of a given substream [22], and is used to predict the direction of future branches in that substream. A branch substream might be defined by some bits of the branch address, by a bit pattern representing previous branch directions (known as *branch history register (BHR*)), by some combination of branch address and branch history, or by bits from target addresses of previous branches [23, 20,19,24,25].

Simple history bit and counter-based prediction schemes [19] achieve prediction accuracies of 85%-90%. However, for today's wide-issue, deeply pipelined processors, a misprediction rate of 10% incurs a severe performance cost. For a four wide processor with twelve stage pipelines, a 10% misprediction rate reduces performance by 55%, from a peak 4 instructions per cycle (IPC) to 1.8 (assuming one branch every four instructions).

Yeh and Patt [19] proposed the two-level adaptive branch predictor which uses one or more k-bit global history registers (BHR), to record the branch outcomes of the most recent k branches. It also uses one or more arrays of predictor tables (PHT), to keep track of the more likely direction for branches. The lower bits of the branch address are used to select the appropriate PHT and the content of the BHR selects the appropriate 2-bit counter to use within that PHT. The two-level branch predictor using 15 global history bits achieves a prediction accuracy of 93.3%. Therefore, two-level dynamic branch predictions have been incorporated in several microprocessors such as Pentium Pro and Alpha 21246.

Several variations of the two-level branch predictor have been proposed by Yeh and Patt[21] which achieves a prediction accuracy of 92.5% with a branch history length of 10. McFarling [24] introduced *gshare*, a variation of the global-history two-level branch predictor which XORs the global branch history with the branch address to index into the PHT. A gshare using 13 history bits achieves a prediction accuracy of approximately 92%.

Due to chip die-area budgets and access-time constraints the size of these dynamic tables is limited, and sometimes two different branches end up sharing the same PHT entry. This is called ***prediction table interference (aliasing)***. As Sechrest, et al. showed [26], aliasing is the main cause for decreased prediction accuracy. Therefore, a variety of techniques for reducing aliasing have been suggested [24,27,28,29]. The gshare predictor with no PHT interference achieves 95% prediction accuracy, leaving a significant room for improvement.

One of the first lookahead branch predictors was presented by Yeh, et al. [31] as the multiple branch two-level adaptive branch predictor. This predictor uses the result of the first branch prediction to speculatively update the history register for a second branch prediction. Onder, Xu, and Gupta propose a similar scheme in which predictions for an entire branch sequence are made all at once, and instruction fetch can continue unimpeded through the last branch [32].

Driesen and Hölze propose a "cascaded" predictor that dynamically filters easily predicted branches, relieving aliasing effects in the PHT [33]. The Alpha 21264 branch predictor uses an idea based on overriding: the branch predictor can override the less accurate instruction cache line predictor, with a penalty of a single cycle, as opposed to the seven-cycle branch misprediction penalty [34].

Lee et al. [27] proposed the ***bi-mode*** branch prediction to improve the predictions by eliminating the aliasing. The bi-mode scheme can outperform other dynamic predictors, yet it can still suffer from interference between the weakly biased and strongly biased substreams.

To further improve prediction accuracy, ***hybrid branch predictors*** have been studied [24,35]. A hybrid branch predictor is composed of two or more single-scheme predictors and a mechanism to select among these predictors. A hybrid branch predictor may achieve a prediction accuracy of 93.9%.

To address the problem of target prediction of indirect branch, Driesen and Holzle [36] explored a large space of path based indirect branch predictors. In their experiments, a global path history was shown to be better than per branch address path histories. They also introduced a hybrid predictor where both components used global path histories but each component used a different length history.

Juan et al. [37] proposed variations of global pattern history based prediction schemes in which the number of pattern history bits used to generate an index into the PHT(s) was not fixed. All predictions made during an interval were made using the number of history bits selected by the hardware at the beginning of the interval. The advantage of these variations is that the number of history bits used dynamically adapts to the code. Stark et al. [38] achieved a substantial improvement over Juan's work by

constructing the index into the predictor table as a function of the last *N* target addresses, and using profiling information to select the proper value of *N* for each branch.

However, recent studies show that as clock rates increase and die-sizes decrease, delays will have an increasingly significant impact on the time to access large structures such as branch prediction tables [39,40]. For instance at a moderate clock rate of 1 GHz, an 8K-entry *gshare* predictor with average accuracy of about 94% can be built. Thus, predictors must become more accurate to support deeper pipelines, but they must also become smaller because of aggressive clocking.

This paper shows performance measured in terms of IPC, prediction accuracy and misprediction rate, exclusively, as a function of instruction-window size, instruction-cache size, machine issue width, and prediction buffer sizes for different schemes of branch predictors. The branch prediction schemes chosen for these comparisons *are statically taken/not-taken*, *bimodal*, *combination*, *correlation*, *two-level adaptive* (TLA), *hybrid*, and *gshare* branch prediction schemes. Metrics for the various schemes was generated with the sim-inorder and sim-outorder simulators from the SimpleScalar tool suite written by Todd Austin [41,42], and the extended version Hydrascalar by Keven Skadron [43]. Implementation of these schemes included various modifications to the bpred.c, bpred.h and sim-inorder.c files.

The remainder of the paper is organized as follows. The next section introduces the ILP processor Mode. Section 3 provides details on the various branch prediction schemes and the related work. Section 4 investigates the implementation cost and performance metrics. The experimental framework is described in section 5. Simulation results are presented starting with section 6, which deals with the influence of size and organization of the branch target buffer. Finally, we conclude the results presented.

II. ILP PROCESSOR MODEL

Current processors [34,44,45] improve performance by using aggressive techniques to exploit ILP. These techniques include multiple instruction issue; out-of-order scheduling; non-blocking caches; and speculative execution. Fig. 1 illustrates the parallel execution method used in most superscalar processors. Instructions begin in a static program. The instruction fetch process, including branch prediction, is used to form a stream of dynamic instructions. These instructions are inspected for dependences with much artificial dependence being removed. The instructions are then dispatched into the window of execution. In Fig. 1, the instructions within the window of execution are no longer represented in sequential order, but are partially ordered by their true data dependences and hardware resource availability. Finally, after execution, instructions are conceptually put back into sequential program order as they are retired and their results update the architected process state.

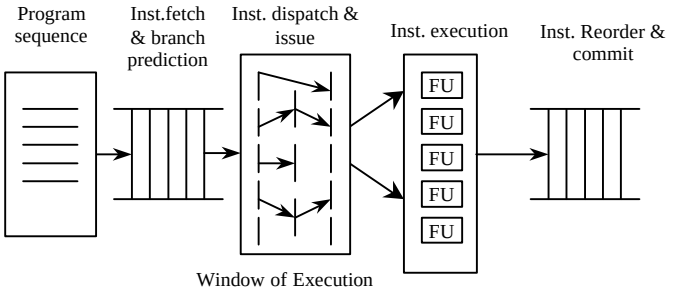


Fig. 1 Traditional Superscalar Architecture

A fast processor requires balancing and tuning of many microarchitectural features that compete for processor die cost and for design and validation efforts. For example, Fig. 2 shows the basic Intel NetBurst™ microarchitecture of the Pentium® 4 processor [44].

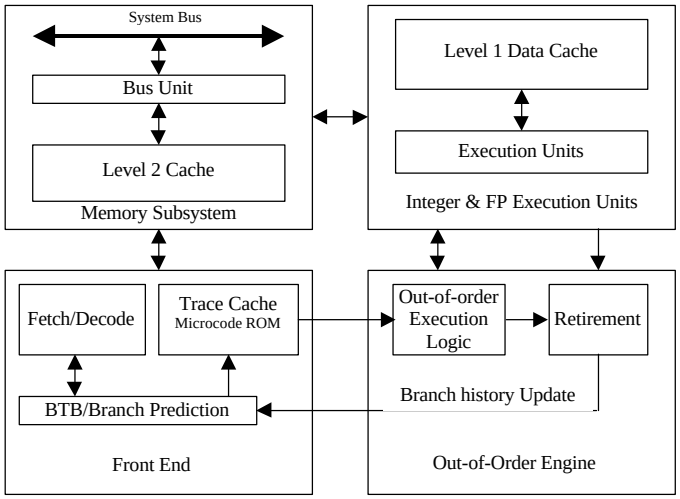


Fig. 2 A Modern Processor Basic block diagram

III. BRANCH PREDICTION SCHEMES

To provide context for our research, we now review some of the recent work in branch prediction. For this study, the following predictors configuration are studied: *static*, *bimodal*, *combinational*, different variations of *two-level adaptive prediction*, *gselect* and *gshare*, *skewed*, and *hybrid*. Most proposed predictor organizations today are either refinements of *two-level* predictors [24,27,46,47], or *hybrid* organizations combining two or more components [24,48,49].

The principal function of any predictor is to take an input, corresponding to the current state of the system, and generate an output that predicts some as yet uncomputed value. The predictor structure may be represented as a primitive as shown in Fig. 3.

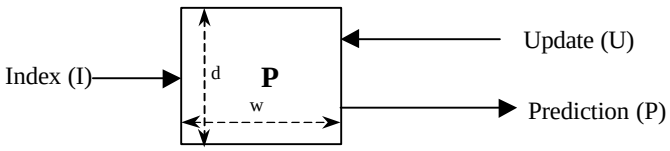


Fig. 3 Primitive Branch Predictor.

This primitive is basically a memory that is *w* bits wide and *d* entries deep. As with a typical memory, it has two operations. For our purpose, however, rather than read and

write, the two operations are called *predict* and *update*, and furthermore these two operations are always used as a pair.

The operation of the predictor consists of a predict step in which the memory is accessed at address index *I* and the value read is used as the prediction *P*. When the prediction resolves an update value *U* is delivered to the predictor and written into the same location indexed by *I*. An extension might involve writing back to a different location or there might be multiple predictions before there is an update.

The primitive predictor in Fig. 3 may be represented as an algebraic expression [50]:

P[w,d](I;U)

where

w = width

d = depth

I = index for prediction and update

U = update value

Branch prediction schemes are divided into two classes: ***static*** and ***dynamic***. In order to explain ***dynamic branch prediction***, one has to differentiate it from ***static branch prediction***.

A. Static Branch Prediction

Static Branch Prediction in general is a prediction that uses information that was gathered before the execution of the program. This prediction was obtained either using very simple heuristics [16], static analysis [17], or profile information [18]. The simplest of these predicts that all conditional branches are always *taken* as in Stanford MIPS-X, or always *not taken* as in Motorola MC88000. This latter scheme is effective for loop intensive code, but does not work well for programs where the branch behavior is irregular. Predicting all branches to be *taken* achieves about 66% accuracy whereas predicting *not taken* achieves about 34% for the SPEC95 integer branches. With additional hint bits in the branch opcodes, PowerPC 604 allows the compiler to pass prediction information to the hardware. The profile-guided branch predictor bases its prediction on the direction the branch most frequently takes, which is determined by profiling the program on a training input data set and presetting a static prediction bit in the opcode. Unfortunately, branch behavior for the sample data may be very different from the data that appears at run-time. The compiler, through profiling or static heuristics can provide hints to the microarchitecture about the likely direction of the branch.

Static branch predictors are usually less accurate than dynamic branch predictors because they cannot consider changing conditions at run-time.

B. Dynamic Branch Prediction Schemes

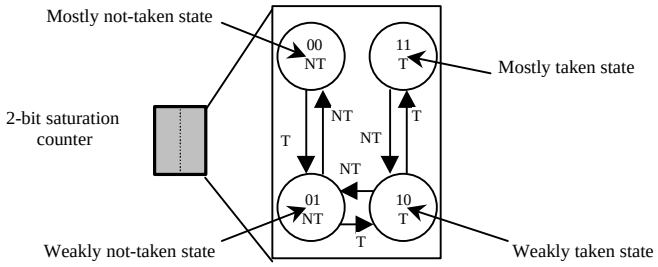
Dynamic branch prediction uses information about taken or not taken branches gathered at run-time to predict the outcome of a branch. There are several dynamic branch predictors in use or being researched nowadays. Those include one-level branch predictors, two-level branch predictors, and multiple component branch predictors.

1) Bimodal Predictor

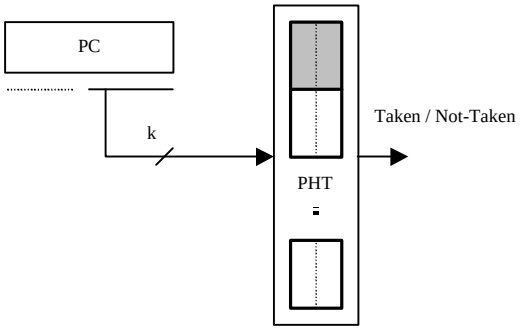
For one-level branch prediction, in the simple case of direct mapping and just the branch address as information source, one takes *k* least significant bits of the branch instruction address in order to map it to an array of a single column table of 2^k entries. Each entry consists in general of a *n*-bit counter, which is designed as a saturated counter in most cases. For example, The ***bimodal*** or "two-bit counter (2bC)" scheme, proposed by Smith [16], contains a table of two-bit saturating binary counters or pattern History table (PHT), which is indexed by the branch address to provide the direction prediction.

Fig. 4 shows the finite state machine for incrementing the 2-bit saturation counters and the bimodal predictor organization respectively. A saturated counter works like this: each time the branch is taken the counter is incremented; if the maximum value is already reached, nothing is done. Each time the branch is not taken the counter is decremented; if the minimum value = 0 is reached it stays the same. Therefore, if the value of the counter is below 2^{n-1}, the branch is predicted as not taken, otherwise the branch is predicted as taken. An 8K 2-bit counter scheme achieves a prediction accuracy of 87%. The two-bit predictor scheme only uses the recent behavior of a single branch to predict the future of that branch.

Pan et al. [51] developed the ***correlation-based branch predictors*** that additionally use the behavior of other branches to make a prediction. While two-bit predictors use self-history only, the correlating predictor uses neighbor history as well. A correlation-based predictor denoted as an (*m*,*n*)-predictor uses the behavior 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 BHR. The contents of the BHR are used to address (index) the entries in a so-called *pattern history table* (PHT). Typically two-bit predictors are used in PHTs. For example, a (2,2)-predictor uses a BHR of two bits to choose among four 1k-entry PHTs. The entries of the 1k-entry PHTs are generally accessed by using the lower order 10 bits of the branch address. The four 1k-entry PHTs can also be viewed as a single 4k-entry PHT. Then 12 bits are required for the PHT lookup. Therefore, two bits from the BHR are concatenated with 10 bits from the branch address. A number of processors, such as the Intel Pentium, Alpha 21164 [52] and the MIPS R10000 [53] use this predictor.



2-bit predictor saturation counter states



Bimodal predictor

Fig. 4 Two-bit saturation counter state diagram and bimodal predictor.

2) Two-level Adaptive Branch Prediction

Yeh and Patt [20,21] proposed the two-level adaptive branch prediction scheme. By tracking branch history, two-level predictors can identify branch behavior patterns that a bimodal, single-level table--one bias counter per branch site-- cannot expose. The history can either ***global***, tracking outcomes from different branches in the same history register, or ***local***, tracking history separately for each branch.

The basic two-level predictor structure shown in Fig. 5 uses a single global BHR of k bits to index a PHT of 2-bit counters. The global BHR is updated with outcomes from all branches. All schemes that use a single global BHR are called ***global history schemes*** and correspond to Pan et al.'s ***correlation-based predictor schemes.***

When a conditional branch B is being predicted, the content of its BHR dented as $R_{c-k}\ R_{c-k+1} \ldots R_{c-1}$, is used to address the PHT. The pattern history bits Sc in the addressed entry $PHT_{Rc-k\ Rc-k+1\ \ldots\ Rc-1}$ in the BHT are then used for predicting the branch. The prediction of the branch is

$$Z_c = \lambda(S_c) \qquad (1)$$

where λ is the prediction decision function.

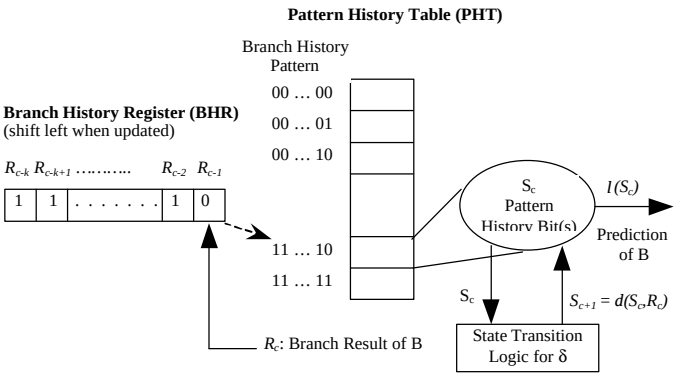


Fig. 5 Structure of a global history two-level branch predictor.

After the conditional branch B is resolved, the outcome R_c is shifted left into the BHR in the least significant bit position and is also used to update the pattern history bits in the PHT entry $PHT_{Rc-k\ Rc-k+1\ \ldots\ Rc-1}$. After being updated, the content of the BHR becomes $R_{c-k+1}\ R_{c-k+2} \ldots R_c$ and the state represented by the pattern history bits becomes S_{c+1}. The transition of the pattern history bits in the PHT entry is done by the state transition function d, which takes in the old pattern history bits, and the outcome of the branch as inputs to generate the new pattern history bits. Therefore, the new pattern history bits S_{c+1} become

$$S_{c+1} = d(S_c, R_c) \qquad (2)$$

A straightforward combinational logic circuit is used to implement the function δ to update the pattern history bits in the entries of the PHT. The transition function d, predicting function l, pattern history bits S and the outcome R of the branch comprises a finite-state More machine, characterized by equations 1 and 2.

In ***local history*** schemes, the processor maintains two tables namely a branch history table (BHT) and PHT. The BHT contains the history of directions taken by recently executed k branches separately, where k is the width of BHT. The PHT is indexed by the BHT entries while BHT itself is indexed by the lower address bits of PC.

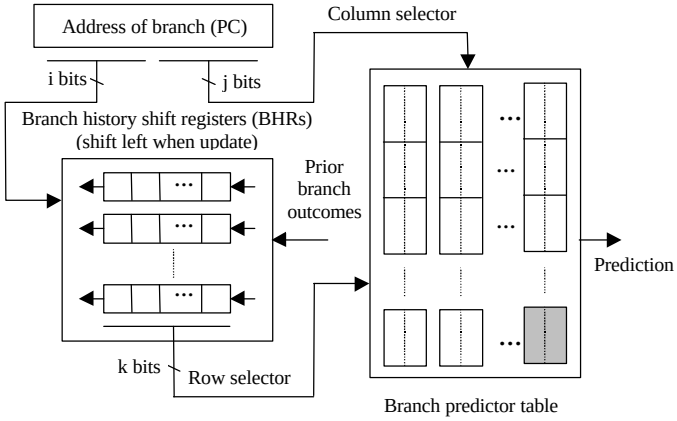


Fig. 6 General Structure of Two-Level Adaptive Branch Prediction.

The general structure of two-level adaptive branch prediction algorithm [21,51] uses one or more k-bit BHR(s) as shown in Fig. 6 to record the outcomes of the most recent k branches. The prediction table is organized as $2^j x 2^k$ 2-bit counters, i.e., one or more PHT(s), to keep track of the more-likely direction for branches. The low-order j bits of the branch's address select the appropriate PHT, while the k-bit value in the BHR selects the appropriate 2-bit counter to use within that PHT.

Two-level adaptive schemes have proven especially effective, because they correlate behavior among different branches. Therefore, they have been implemented in at least two main stream high-performance processors, the Pentium-Pro, Pentium II [54] and the Alpha 21264 [55].

Yeh and Patt [21] classified the two-level adaptive branch prediction model into three classes according to the way the first-level branch history is collected. They are characterized as follows:

a) Global History Schemes (GAg, GAs, GAp)

Also called *Correlation Branch Prediction* by Pan et al. [51], the first-level branch history means the actual last k branches encountered; therefore, only a single global history register (GHR) is used. The GHR is updated with the results from all the branches. Since the GHR is used by

all branch predictions, not only the history of a branch but also the history of other branches influences the prediction of the branch.

b) Per-address History Schemes (PAg, PAs, PAp)

In the per-address history schemes, the first-level branch history refers to the last *k* occurrences of the same branch instruction; therefore, one history register is associated with each static conditional branch to distinguish the branch history information of each branch. One such history register is called a per-address history register (PBHR). The per-address history registers are contained in a per-address branch history table (PBHT) in which each entry is indexed by the static branch instruction address. In these schemes, only the execution history of the branch itself has an effect on its prediction; therefore, the branch prediction is independent of other branches' execution history.

c) Per-set History Schemes (SAg, SAs, SAp)

In the Per-set history schemes, the first-level branch history means the last *k* occurrences of the branch instructions from the same subset; therefore, one history register is associated with a set of conditional branches. One such history register is called a per-set history register. The set attribute of a branch can be determined by the branch opcode, branch class assigned by a compiler, or branch address. Since the per-set history register is updated with history possibly from all the branches in the same set, the prediction of a branch is influenced by other branches in the same set. The full table of Yeh and Patt's two-level adaptive branch predictors is given as shown in table 1 [51]:

TABLE 1 THE NINE VARIATIONS OF TWO-LEVEL ADAPTIVE BRANCH PREDICTION

	Global PHT	Per-set PHTs	Per-address PHTs
Global BHR	GAg	GAs	GAp
Per-address BHT	PAg	PAs	PAp
Per-set BHT	SAg	SAs	SAp

(G, P, and S referred to BHR; g, p, and s referred to PHT; A stands for "adaptive")

3) Gselect and gshare predictors

McFarling [24] analyzed the two-bit predictors and correlation predictor schemes and introduced a number of new predictors. One set of new correlation-based predictors' uses a *hash function* into PHT instead of indexing the PHT to reduce conflicts. Pan et al. [19] strategy ***gselect l/l*** closely resembles global prediction technique except that the concatenation of the lower order *l*-bits of the branch address (PC) with the lower *l*-bits of the BHR to index into the PHT. Performance of *gselect* can vary depending upon the number of bits distributed for each of BHR and PC during concatenation. *Gselect* uses *k/2* bits of PC and *k/2* bits of BHT to form index into PHT with 2^k entries. *k/2* bits may not be enough to capture either local or global branch patterns.

In contrast to simple indexing, McFarlings's [24] ***gshare k/k*** technique uses the bitwise XOR of all *k*-bits of both the branch address and the BHR as a hash function to index into the PHT as illustrated in Fig. 7.

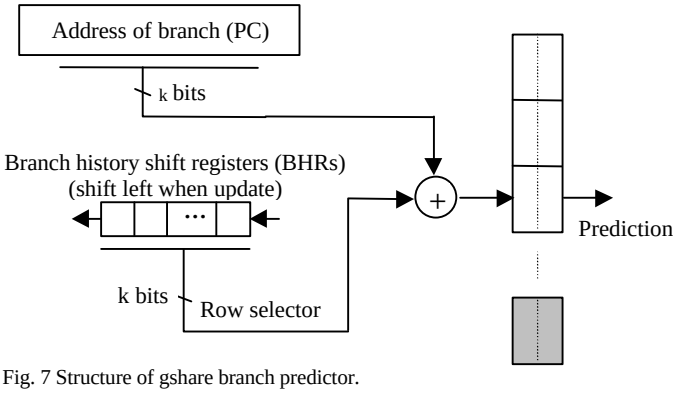


Fig. 7 Structure of gshare branch predictor.

Gshare generally shows improvement over gselect as it uses more bits of BHR as well as PC. XOR hashing helps in distinguishing different and unique branch patterns by distributing the counters evenly.

4) Skewed Branch Predictor

Michaud et al. [56] introduced ***the Skewed branch predictor***, a multi-bank, tag-less branch predictor, which is designed specifically to reduce the impact of conflict aliasing. A skewed branch predictor is constructed from an odd number (typically 3 or 5) of predictor-table banks, each of which functions like a standard tagless predictor table. When performing a prediction, each bank is accessed in parallel but with a different indexing function and a majority vote between the resulting lookups is used to predict the direction of the branch. It is like correlation-based scheme, which uses the global history. It takes global history (BHR) bits and the address of the branch instruction (PC), and uses this information to index three branch counter tables. Three different hash functions are used to index the tables (each hash function indexes its own table). A majority vote is taken for final prediction by comparing 2bCs read from three counter tables. Michaud et al's showed that a good choice of hash functions can eliminate almost all aliasing.

5) Hybrid Predictors

A *hybrid predictor* includes multiple predictors [24,48], with some way to choose which predictor to use at any time. Typically, two main predictors and a selector predictor that chooses the better of the two main predictors to predict a given branch. In principle, all kinds of branch predictors are candidates for combination of predictors. McFarling combined the *2-bit* predictor [24] 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 prediction as predicted by the 2-bit predictor can be used.

In a hybrid organization like the one in Fig. 8, the components are themselves often two-level components; in this way, the overall predictor can make both global and local history available to the program.

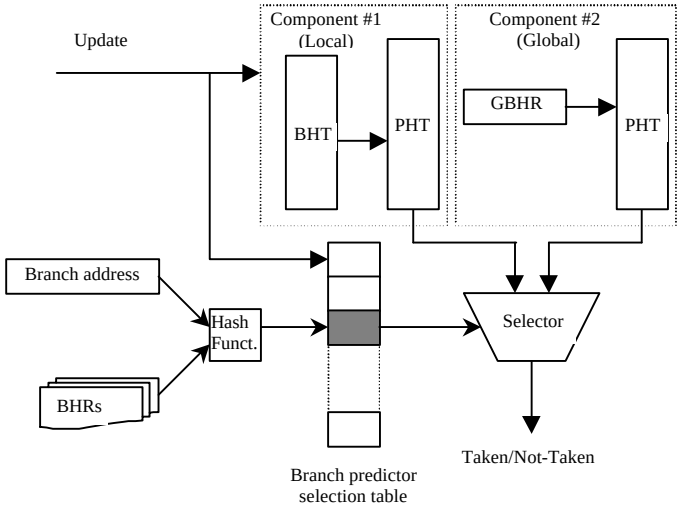


Fig. 8 The organization of a hybrid predictor with two component.

Grunwald et al. [57] used the SAg and gshare and showed that the processor typically issued 20-100% more instructions than actually commit, due to speculative execution. The selection mechanism can be static, with branches assigned to a particular component at compile time, or dynamic, with a hardware structure that learns when to use the different components. The dynamic selector, instead of tracking branch outcomes, tracks predictor success.

IV. Implementation Cost & Performance Metrics

A. Performance Cost

The implementation cost models for the single scheme dynamic predictors are shown in Table 2. Where '*j*' is the number of PHTs, '*i*' is the number of PC bits for indexing the BHT, and '*k*' is the number of bits of the BHR.

TABLE 2 HARDWARE COSTS FOR THE SINGLE-SCHEME DYNAMIC PREDICTORS

Scheme Name	*i*	*j*	*k*	Buffer size (bits)
Bimodal	*Var.*	*N/a*	*N/a*	$2x2^i$
Correlation	*1*	*Var.*	*2*	$2+2x2^kx2^i$
gselect	*1*	*Var.*	*Var.*	$k+2x2^{(i+k)}$
Global	*1*	*1*	*Var.*	$k+2x2^k$
Local	*Var.*	*Var.*	*Var.*	$kx2^i+2x2^{(i+k)}$
gshare	*N/a*	*Var.*	*Var.*	$k+2x2^k$
GAg(k)	*N/a*	*1*	*Var.*	$k + 2^k x 2$
GAs(k,p)	*N/a*	*p*	*Var.*	$k + p x 2^k x 2$
GAp(k)	*N/a*	*b*	*Var.*	$k + b x 2^k x 2$
PAg(k)	*Var.*	*1*	*Var.*	$b x k + 2^k x 2$
PAs(k,p)	*Var.*	*p*	*Var.*	$b x k + p x 2^k x 2$
PAp(k)	*Var.*	*b*	*Var.*	$b x k + b x 2^k x 2$
SAg(k)	*Var.*	*s x 1*	*Var.*	$s x k + s x 2^k x 2$
SAs(k,sxp)	*Var.*	*s x p*	*Var.*	$s x k + s x p x 2^k x2$
SAp(k)	*Var.*	*s x b*	*Var.*	$s x k + s x b x 2^k x2$

k history register length *p number of PHTs*
b is the number of BHT entries *s* is the number of branch sets.

Comparing the cost effectiveness of the different schemes using the formulas in Table 2 requires either approximately the same accuracy or a fixed hardware budget. Fig. 9 shows the accuracy vs. an estimated hardware cost for each predictor type.

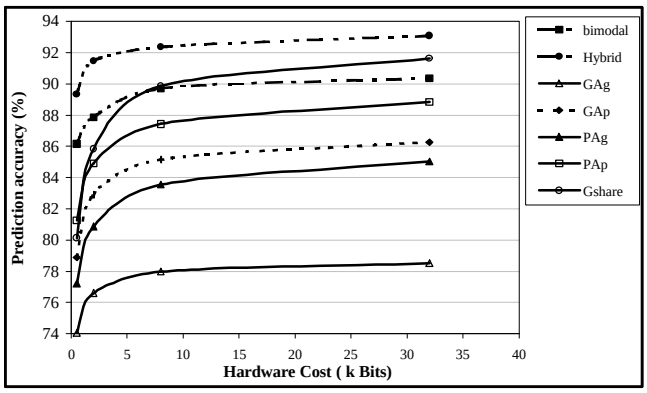


Fig. 9 Prediction Accuracy vs. Hardware Cost.

We notice that the size of the PHT increases exponentially as a function of the BHR length and therefore the cost rises exponentially as a function of BHR length. Yeh and Pat [59] shows, for a fixed hardware budget of 8K, that 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(6,4x16) scores best. When given a higher hardware budget of 128k 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(9,4x32).

Evaluating the predictors scheme when they predict with approximately the same accuracy, for example 97%, GAg requires an 18-bit history register, PAg requires 12-bit history registers, and PAp requires 6-bit history registers. Hence, PAg is the cheapest, GAg's pattern history table is expensive when a long history register is used. PAp is expensive due to the required multiple pattern history tables. For a given hardware cost, the hybrid predictor configuration specifies the classes of the single-scheme predictors used and the amount of hardware devoted to each scheme.

B. Prediction accuracy

In this paper, we use prediction accuracy as the metric to evaluate the performance of branch predictors. The prediction accuracy is the percentage of correctly predicted conditional branches. If we ignore stalls such as cache misses and bus conflicts, the branch penalty is defined as [15]:

$$\text{Branch penalty} = c \times ((1-a) \times q \times IPC) \qquad (3)$$

Where *c* denotes the number of cycles wasted due to a branch misprediction, *a* denotes the prediction accuracy, *q* denotes the ratio of the number of branches over the number of total instructions, and *IPC* denotes the average number of instructions that are executed per cycle. For b=5 and $q \times IPC$ = 0.9, a branch penalty of less than 10% requires a prediction accuracy *a* of greater than 97.7%.

The translation from prediction accuracy to machine performance is not direct. However, higher prediction accuracy means machine pipelines stall less frequently for incorrect branch predictions.

C. Predictor Delay vs. Accuracy

Branch prediction accuracy increases with the amount of memory allocated to the branch prediction table. However, larger structures lead to larger access delays; worse, aggressively increasing clock rates (as the marketplace

demands) increases the structure access time as measured in clock cycles. The cost of executing a branch instruction is roughly approximated by:

$$Cost = t + (r \times y) \quad (4)$$

Where t is the delay of branch predictor, r is the misprediction rate, and y is the misprediction penalty. Jiménez et al. [40] studied the impact of delay on the design of branch predictors.

V. Experimental Framework

This section describes the simulation environment and the benchmarks used in this paper.

A. Simulation Environment

We use the SimpleScalar [2,5,7] execution-driven simulation infrastructure to compare the different branch predictors. Our simulator is an extension of the *sim-outorder* , the SimpleScalar cycle-level superscalar timing simulator. SimpleScalar provides a simulation environment for modern out-of-order processors with speculative execution. The environment enhanced a toolbox of simulation components—like a branch-predictor module, a cache module, and a statistics-gathering module—as well as several simulators built from these components. Each simulator interprets executables compiled by gcc version 2.6.3 for a virtual instruction set, the Portable Instruction Set Architecture (PISA), that most closely resembles MIPS IV [58]. The simulators emulate the executable's execution in varying levels of details.

The simulated processor contains a unified active instruction list, issue queue, and rename register file in one unit called the reservation update unit (RUU). Separate banks of 32 integer and floating point registers make up the architected register file and are only written on commit.

Table 3 summarizes the important features of the simulated processor. The baseline configuration parameters are roughly those of a modern out-of-order processor, which loosely resembles the configuration of an Alpha 21264 [34]. The extended SimpleScalar models in detail an out-of-order execution (OOE), 5-stage pipeline: fetch (including branch prediction), decode (including register renaming), issue, writeback, and commit We add three further stages between decode and issue to simulate time spent renaming and enqueuing instructions. Issue selects the oldest ready instructions.

Like a real processor, the extended SimpleScalar checkpoints appropriate state as it encounters branches and, then, upon detecting a mispredicted branch, wrong-path instructions are squashed and the correct state recovered using the checkpointed state. Modeling the actual instruction flow on misspeculated paths captures consequences like prefetching and cache pollution.

TABLE 3 BASELINE CONFIGURATION OF SIMULATED PROCESSOR

Parameter	Value
Processor core	
Window size	80 inst.
LSQ size	40
Fetch Queue Size	8 inst.
Fetch width	4 inst./cycle
Decode width	4 inst./cycle
Issue width	4 inst./cycle (out-of-order)
Commit width	4 inst./cycle (in order)
Functional units	3 Int. ALUs, 1 FP ALU, 1 int. MUL/DIV, 2 Mem. Ports.
Branch prediction	
Branch predictor	Bimodal: 2K predictor GAg: 10-bit history, 1K 2-bit predictor GAp: 8-bit history, 1K 2-bit predictor PAg: 8x10-bit BHT, 1K 2-bit predictor PAp: 8x10-bit BHT, 64K predictor gshare: 10-bit history, 1k 2-bit predictor Hybrid: 4K x 12 global-history selector 1K x 10-bit GAg predictor 2K Bimodal
BTB	4096 entries, 4 sets
RAS	64 entries
Mispredict penalty	2 cycles
Memory Hierarchy	
L1 data-cache	64K, 2-way (LRU), 32B blocks, 1 cycle latency
L1 inst. cache	64K, 2-way (LRU), 32B blocks, 1 cycle latency
L2	Unified, 8M, 4-way (LRU), 32B blocks, 12-cycle latency
Memory	100 cycle latency
TLBs	128 entry, fully associative, 30-cycle miss latency

The extended SimpleScalar uses a McFarling-style hybrid branch predictor [24] that combines two 2-level prediction mechanisms [59] with a selector that chooses between them. The two components are chosen from the available predictor models. For each predictor, the selector chooses the component most likely to be correct by consulting its own 4K-entry table of saturating 2-bit counters, indexed by local history [35]. Since many PHT entries correspond to not-taken branches, a BTB entry is only allocated for taken branches, permitting the BTB to have a fewer entries than the PHT [57]. A return-address stack is a reasonable solution for implementing indirect jumps.

B. Benchmarks and branch statistics

1) Benchmarks

We conducted this study through simulation driven by traces gathered for eight of the less predictable SPECint95 benchmarks, *compress*, *gcc*, *go*, *ijpeg*, *li*, *m88ksim*, *perl* and *vertex* [60]. We chose the SPECint95 benchmarks because of their relatively complex control flow behavior, which is particularly so for *go* and *gcc*, and reflect a variety of prediction accuracies. We did not simulate the SPECfp95 since those programs typically pose few difficulties for branch predictors.

The selected benchmarks for this study, the input file, the number of instructions issued, the percentage of conditional and unconditional branches, load and stores committed, as well as the percentage of arithmetic instructions are shown in Table 4.

The benchmarks may be compiled using the default optimization flags of the SPEC distribution, and hence, run to completion. We simulated the first 40M instructions of each benchmark using a reduced input data set to exercise more benchmark code.

Considered statically, the SPECint95 benchmarks seem like a reasonable set of programs with which to study branch prediction. All contain a large number of conditional branch instructions, containing over thousands as depicted in Table 4. When the dynamic frequency of these branches is considered, however, potential problems with

benchmarks appear. A higher number of branches do not necessarily lead directly to greater difficulties for predictors. Large proportions of the branches, both for the applications and for the operating system, are very highly biased; they are either usually or almost never taken. These branches include loops, error, bounds checking, and other routine conditionals.

TABLE 4 SPECINT95 BENCHMARKS INFORMATION

Benchmark	Input file	Total Number of Insts.	% of Uncond. Branch Insts.	% of Cond. Branch Insts.	% of Load/Store Insts.	% of Arithmetic Insts.
Compress	Test.in	36 M	6.31	10.94	37.43	45.31
	Bigtest.in	413 M	6.87	12.05	33.91	47.16
	Bigtest.in	4337 M	6.44	14.12	32.19	47.25
Gcc	Genoutput.i	40 M	4.69	15.55	39.77	39.99
	Genoutput.i	86 M	4.69	15.46	39.96	39.89
	Gcc.i	201 M	4.67	15.39	40.38	39.55
	Cccp.i	1263 M	4.47	15.41	40.54	39.57
	Varasm.i	251 M	4.83	15.23	40.80	39.14
Go	2stone9.in	40 M	3.13	11.91	28.44	56.52
	2stone9.in	548 M	3.32	11.33	28.68	56.67
Ijpeg	Vigo.ppm	40 M	2.03	6.86	26.30	64.86
	Vigo.ppm	1465 M	1.43	6.71	25.52	66.35
Li	Train.lsp	40 M	9.36	13.42	43.26	33.96
	Train.lsp	183 M	9.58	13.22	42.45	34.74
	Test.lsp	957 M	8.13	15.45	47.58	28.84
M88ksim	Ctl.in	40 M	9.10	13.63	29.49	54.39
	Ctl.in	245 M	9.21	13.57	22.86	54.36
	Ctl.in	48 M	9.10	13.63	23.27	54.01
Perl	Scrabbl.in	40 M	6.04	13.14	45.70	35.12
	Scrabbl.in	40 M	6.04	13.14	45.71	35.11
Vortex	Vortex.in	40 M	4.19	11.79	53.05	30.95
	Vortex.in	2520 M	4.52	11.89	52.77	30.82

2) Branch Statistics

The frequency of encountering branch instructions is a critical factor that decides the performance of ILP processors. If the frequency is low, basic blocks will be larger and more instruction-level parallelism can be exploited and vice-versa. Instruction traces, including both correctly speculated and misspeculated instructions are generated by the SimpleScalar simulator [41]. Eight integer SPECint95 benchmarks chosen to reflect a variety of prediction accuracies were simulated to completion.

It has been found that on the average, 15-30% instructions in a typical application are branch instructions. In addition, approximately 80% of these instructions are conditional branches. Fig. 10 and Fig. 11 show the instruction classification and branch instructions distribution in SPECint95 benchmarks, respectively.

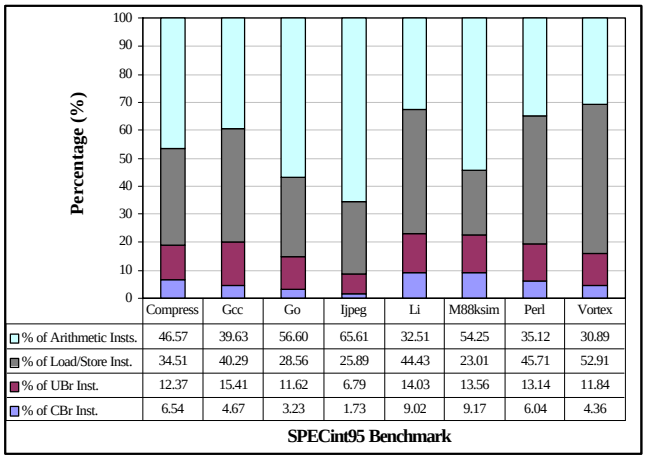


Fig. 10 Instruction Classification.

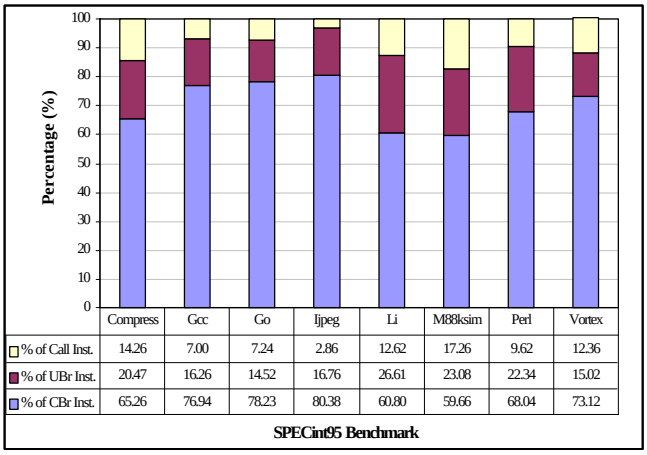


Fig. 11 Branch Distribution; UBr: Unconditional Branch; CBr: Conditional Branch.

VI. SIMULATION RESULTS

In this section we quantify the performance of a selected set of branch predictors and compare between their performance and implementation cost in bits. We used a set of eight programs of the SPECint95 benchmarks for our performance evaluation.

To evaluate the performance of the branch prediction schemes we studied the effects of the branch history lengths ranging from 2 to 18 bits with the number of pattern history tables ranging from 1 to 1024 on their prediction accuracy.

A. Performance Impact of Branch Misprediction

We compared the selected set of branch predictors using the SPECint95 benchmarks. The results are reported in Table 5. The table shows that for SPECint95 benchmark programs about every (sixth) instruction of the trace (the executed and committed instructions) is a branch instruction. The table also shows the branch misprediction rates and the average over all eight benchmarks for seven predictors, bimodal, hybrid, global history schemes, per-address history schemes, and gshare. The PAp and hybrid predictors perform best with 5.98% and 7.28% mispredictions respectively.

Table 6 and Fig. 12 shows the prediction accuracy of the above mentioned seven selected predictors in addition to the static predictors. From which, we notice that hybrid,

bimodal and PAp predictors have prediction accuracy above 95%.

TABLE 5 COMPARISON OF BRANCH PREDICTORS

Application	Committed Inst.		All Inst.		Misprediction Rate						
	No. of Inst.	Branch Inst	No. of Inst.	Branch Inst	Bimodal	Hybrid	GAg	GAp	PAg	PAp	Gshare
Compress	35818648	6179486	53878284	10110157	10.24%	8.19%	22.15%	16.74%	12.00%	5.96%	8.94%
Gcc	27530257	5396831	40000000	7848212	10.94%	9.65%	25.94%	21.11%	22.77%	7.08%	19.87%
Go	25624878	3933606	40000002	6219295	16.89%	16.80%	28.94%	27.28%	27.56%	15.15%	27.20%
Ijpeg	34288308	2985917	40000000	3863854	6.63%	6.48%	21.44%	19.04%	17.93%	6.24%	9.73%
Li	26976059	6156255	40000002	9025523	7.32%	5.64%	15.95%	9.99%	12.54%	5.94%	7.86%
M88ksim	37375040	7603929	50000000	13722410	2.83%	2.81%	15.95%	15.87%	5.57%	2.80%	2.84%
Perl	40482981	7765233	59920964	11640348	6.71%	5.88%	22.46%	17.19%	19.32%	2.61%	15.75%
Vortex	46675933	7457877	50000001	8095335	2.99%	2.76%	25.59%	17.37%	18.39%	2.05%	12.92%
Mean	**34346513**	**5934892**	**46724907**	**8815642**	**8.07%**	**7.28%**	**22.30%**	**18.07%**	**17.01%**	**5.98%**	**13.14%**

Table 6 Branch Prediction Accuracy(%)

Branch Pred. Schemes	Benchmark							
	Compress	**Go**	**Perl**	**Li**	**Gcc**	**Ijpeg**	**M88ksim**	**Vortex**
Static	28.13	34.32	34.32	34.24	37.00	21.02	21.73	34.44
Bimodal	89.76	93.29	93.29	92.67	89.07	93.37	97.17	97.01
Hybrid	91.81	94.18	94.18	94.38	90.42	93.53	97.19	97.27
GAg	77.90	77.55	77.55	84.02	74.22	78.57	84.05	74.51
GAp	83.38	82.83	82.83	90.09	79.04	80.99	84.13	82.65
PAg	88.00	80.68	80.68	87.46	77.28	82.07	94.43	81.65
PAp	94.03	97.40	97.40	94.05	92.94	93.77	97.20	97.96
Gshare	91.36	84.22	84.22	92.18	80.22	90.30	97.16	87.09

- CONSTANT INSTRUCTION CACHE SIZE (IL1) = 64K
- CONSTANT RUU SIZE = 256

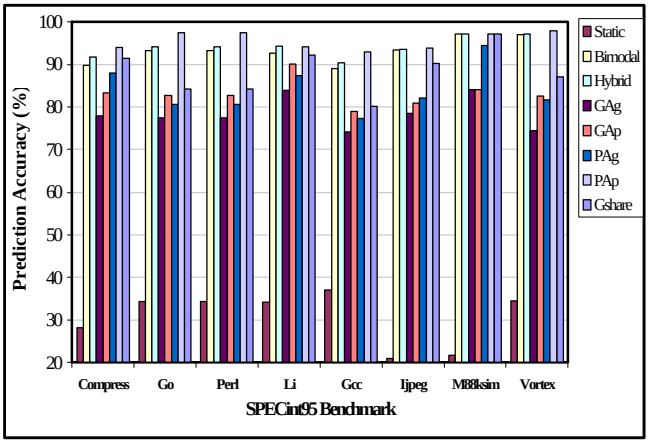


Fig. 12 Prediction Accuracy vs. Benchmarks

Simulation experiments showed that the processor typically issued 14.3%-33.5% more instructions than actually commit, due to speculative execution. The speculative execution factor, given by:

Speculative execution factor = the number of instruction decoded / the number of instruction committed

is 1.36.

B. Performance Impact of Instruction Window Size(RUU)

The baseline superscalar implementation models squash all instructions after a mispredicted branch, thereby limiting the effective window size (RUU). Following a squash, the window is often empty and several cycles are required to re-fill it before instruction issuing proceeds at full efficiency. Furthermore, we are fast approaching the point where the hardware window that can be constructed exceeds the average number of instructions between mispredictions.

The performance influence of *instruction window size* on conditional branches for seven predictors is shown in Fig. 13. Performance is measured in terms of the average number of instructions completed per cycle (IPC) and is shown as a function of window size for fixed instruction cache size of 64K. Our step sizes were chosen to yield a feasible number of simulations spanning a reasonable range of sizes. The IPC scale varies among benchmarks to best show the details. Most benchmarks show a plateau in performance at 48-64 window entries. The results show that, for many integer codes, further increase in window sizes are unwarranted unless coupled with concomitant increases in branch-prediction accuracy. In fact, the predictors of the largest accuracies, such as hybrid, PAp, and bimodal, get the most benefit, for most of the benchmarks, from additional RUU. This is due to the fact that, as prediction accuracy improves, bigger RUUs are more likely to be filled with correctly speculated instructions.

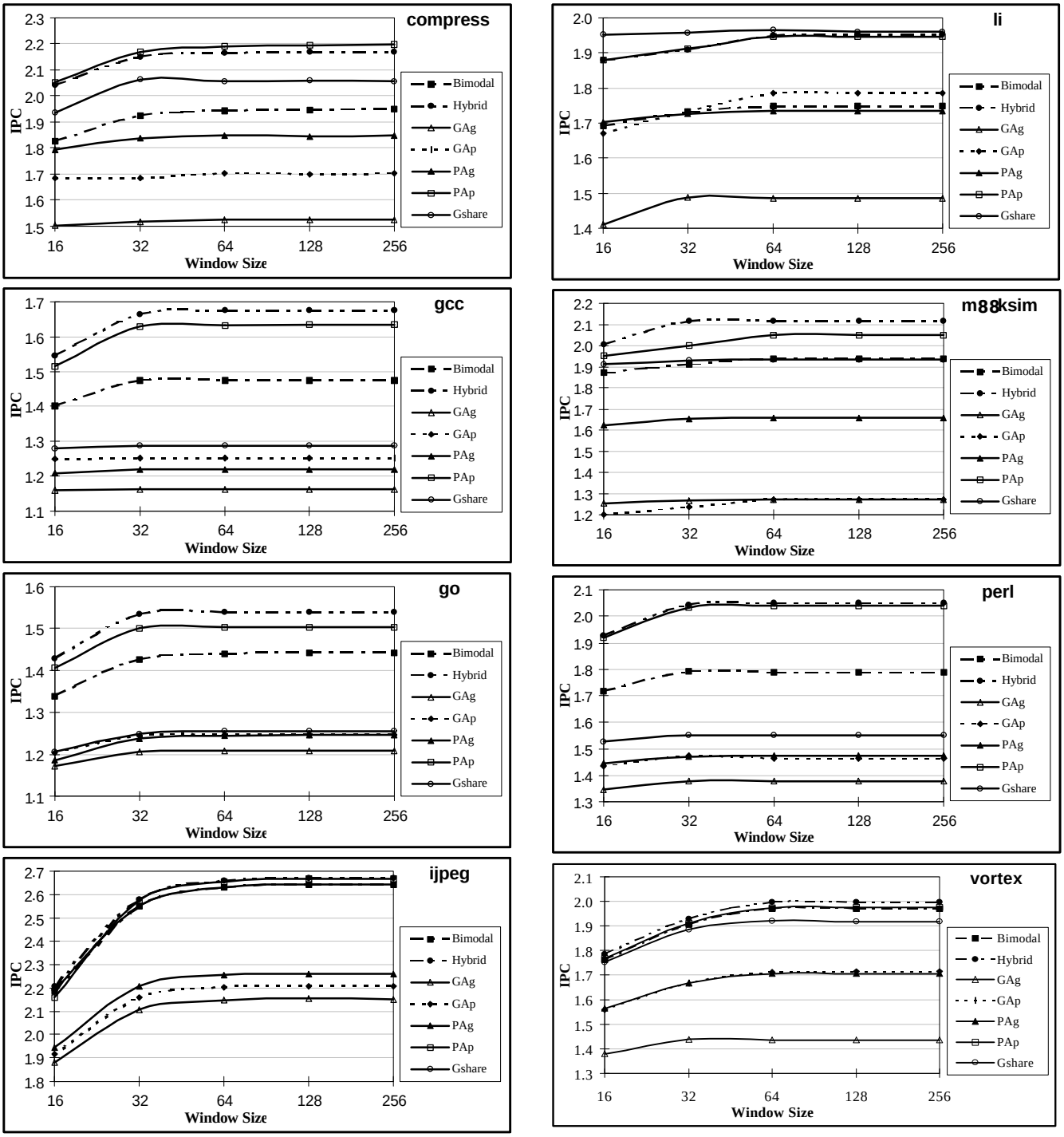


Fig. 13 Performance (IPC) vs. Window Size for Seven Various Branch Prediction Techniques (Instruction Cache Size = 64KB).

The relative performance improvement of, for example, *hybrid* predictor over *gshare* for each of the window sizes is summarized in Fig. 14.

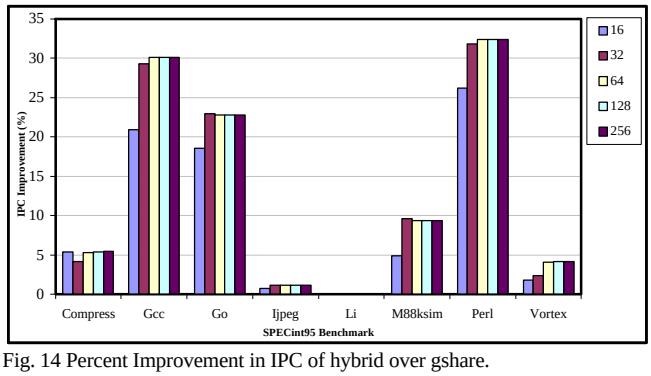


Fig. 14 Percent Improvement in IPC of hybrid over gshare.

C. *Performance Impact of Machine Width*

The performance of the baseline processor model of different widths (the maximum fetch bandwidth, e.g., 4-wide 8-wide, and 16-wide), with various prediction schemes are shown in Fig. 15. As expected, most benchmarks show a plateau in performance at 8-12 machine width. The results show that, for almost all the benchmarks, further increase in machine width above 16 does not affect the branch-prediction accuracy. The performance of the predictors of the least accuracies, such as GAg, PAg, and GAp, rise more slowly with machine width than the other predictors of the largest accuracies because of their higher branch misprediction penalties.

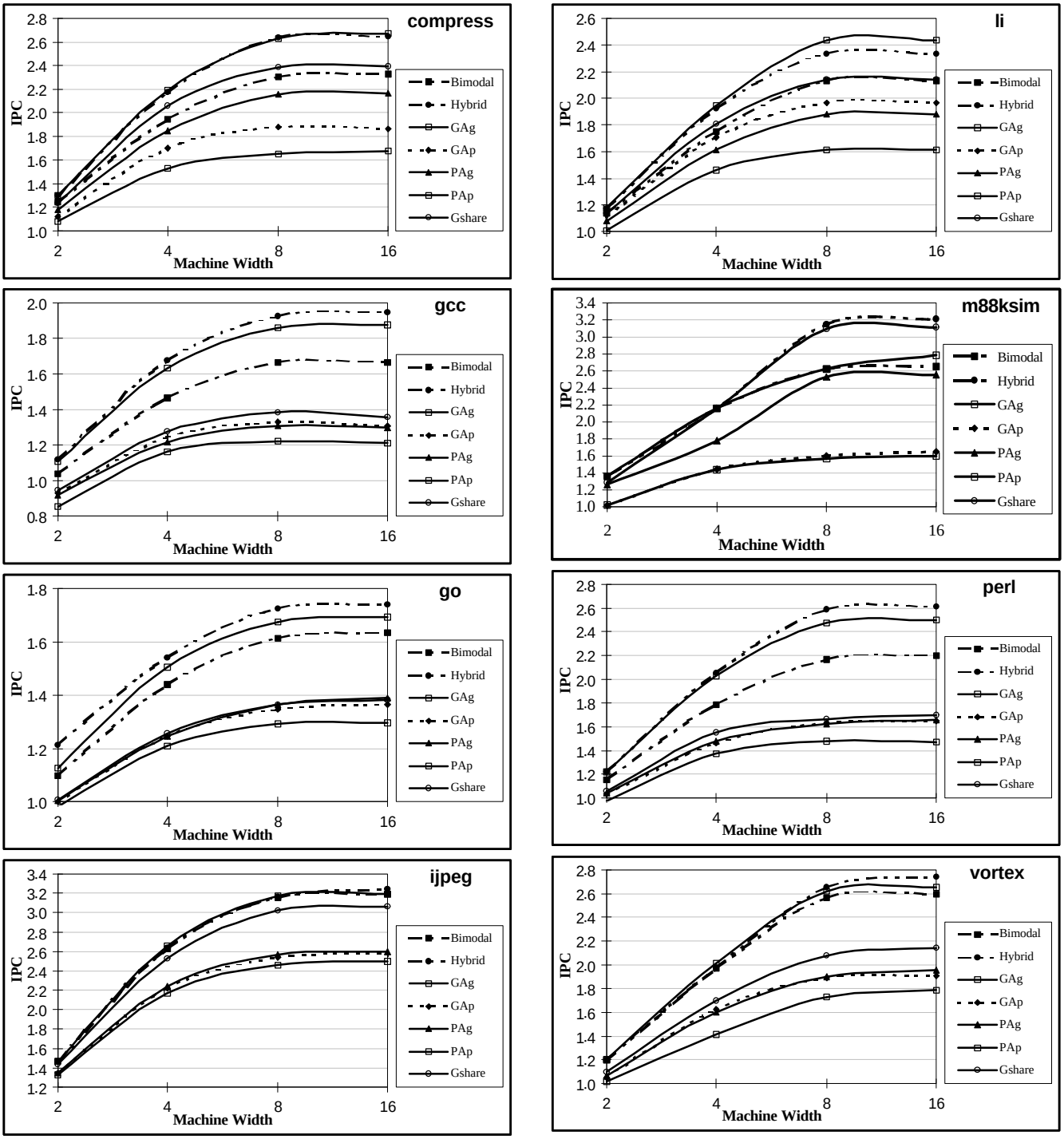


Fig. 15 Performance vs. Machine Width for Various Branch Predictors (Instruction Cache Size = 64KB).

D. Effects of Predictor Size

Predictor sizes from 512 bytes to 16KB are explored for the examined single-scheme and hybrid branch predictors for the eight benchmarks as shown in Fig. 16. Clearly, the prediction accuracy of the examined branch predictor increases with increasing the predictor size. The larger the table, the less interference, but the access time is also increases, especially for higher clock rates. Fig. 16 shows that, for most benchmarks, gshare and GAp always achieving the most benefit from additional predictor sizes. For *gcc* and *go* benchmarks the predictors, in general, achieved lower accuracy rates because they contain a large number of branches in its working set. This large set can cause interference in the pattern history tables of the gshare and the two level predictors, reducing their ability to make accurate predictions. In addition, the large number of branches can incur a significant training cost. The gshare predictor must train itself on the first few instances of the branch before it can begin to accurately predict it. Fig. 15 shows, for most benchmarks except gcc and go, that hybrid, PAp and gshare achieves greater improvements for the smaller configurations. For *compress* benchmark gshare is outperforms most of the individual predictors examined.

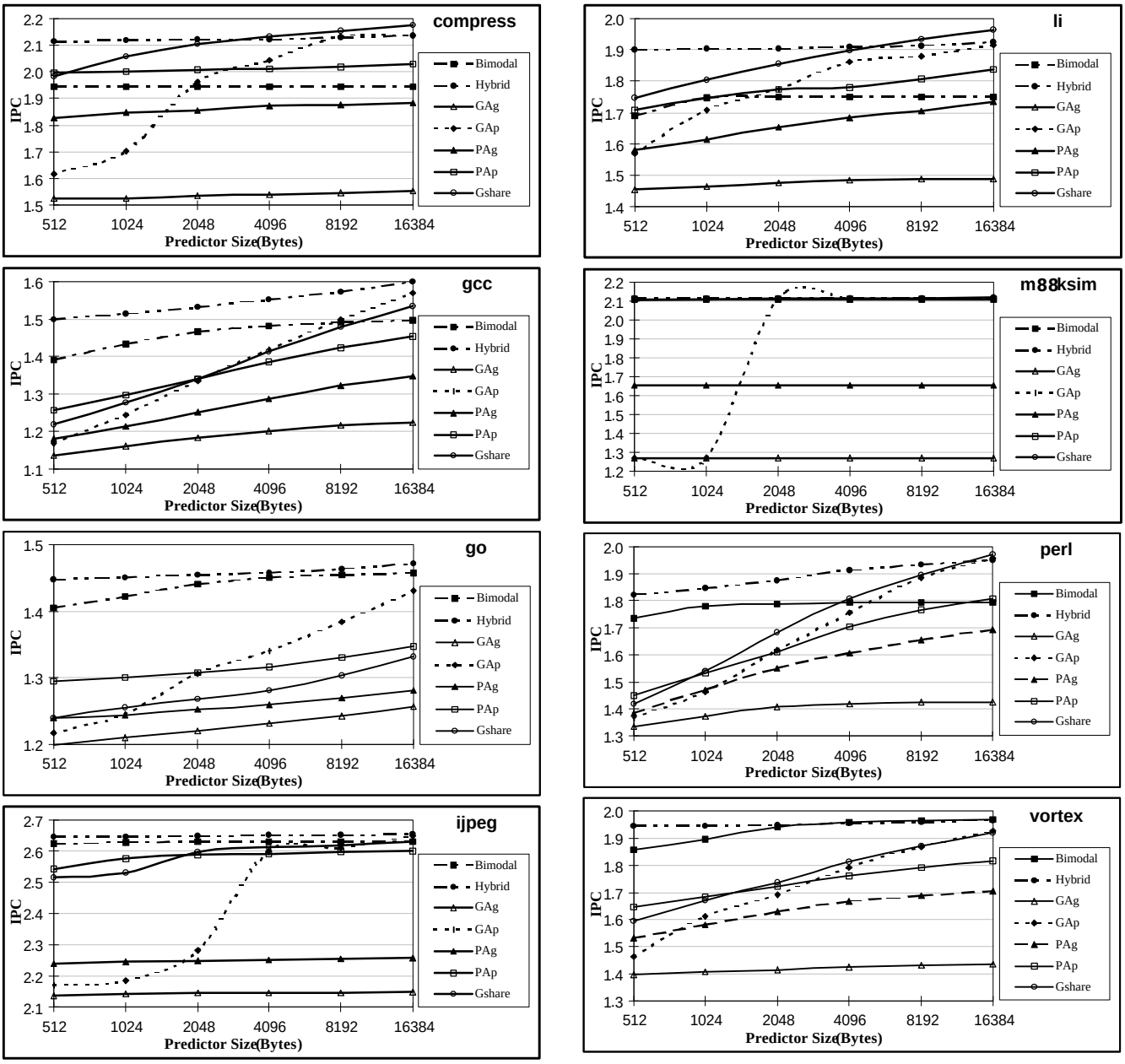


Fig. 16 Performance vs. Predictor Size (Instruction Cache Size = 64KB).

Prediction table interference are classified into three groups [63]: neutral interference when the conflict does not change the prediction, and positive or negative if the conflict changes the prediction for good or bad. There is a significant reduction in the number of negative conflicts but larger amount of neutral interference as the predictor configuration increases. In fact, the increase in the number of not taken branches favors positive interference, because it is more likely that when two branches interfere, they both behave the same way (both not taken) resulting in a positive or neutral conflict. On the other hand, the high faction of not taken branches may be hindering the branch distribution in the BHR. If BHR will tend to be full of zeros, causing much possible BHR values to be never or rarely used, leading to a worse branch distribution and a loss of useful information to make a correct prediction.

E. Performance Impact of Instruction Cache

Navarro et al. [61] and Skadron et al. [62] have shown that the performance impact of the branch predictor is heavily dependent of the instruction cache performance. The performance benefits of an instruction cache miss reduction could compensate for the performance loss due to reduced branch prediction accuracy.

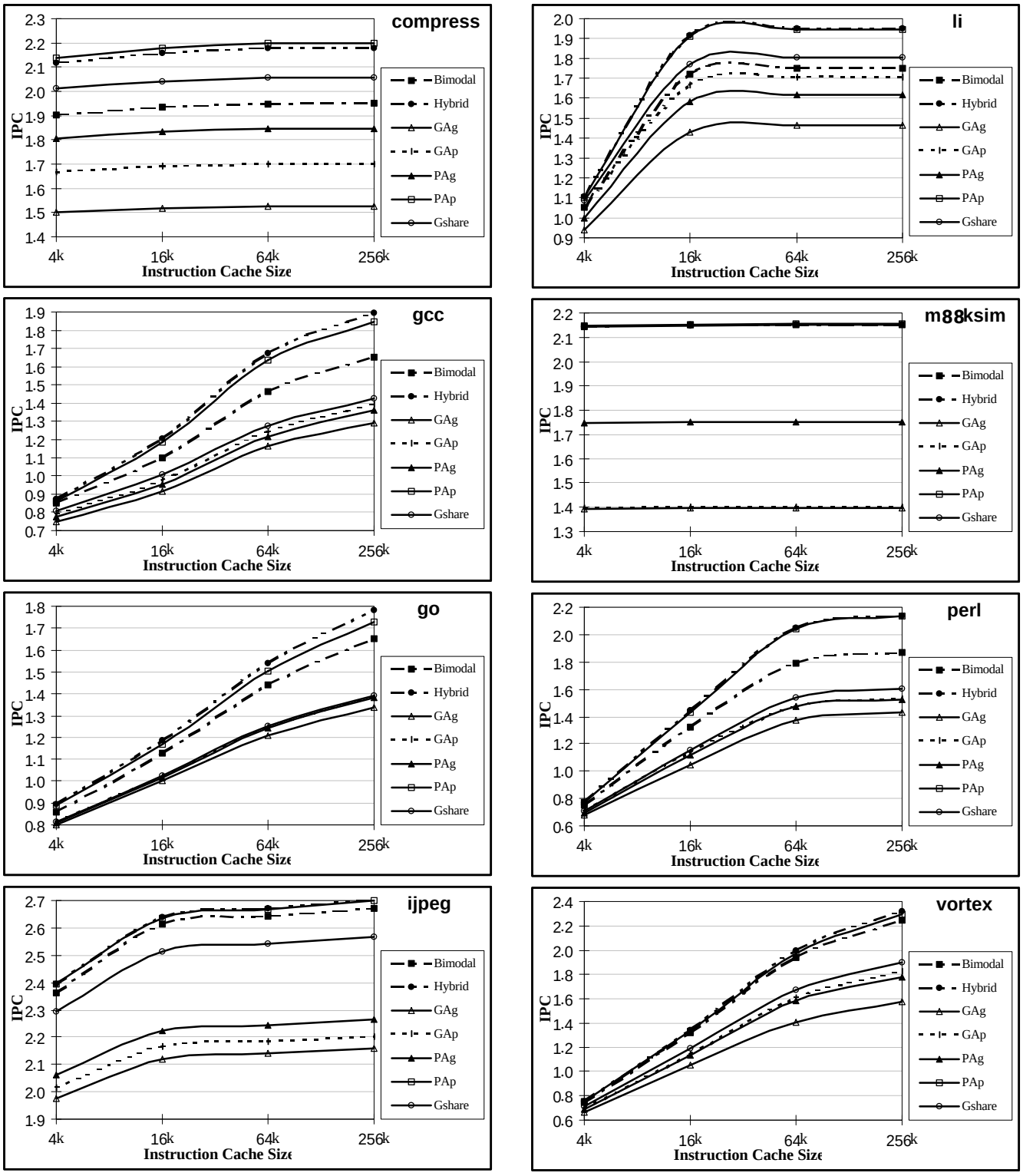


Fig. 17 Processor Performance Measured in IPC for Predictors vs. Instruction Cache (RUU = 256).

Fig. 17 shows processor performance measured in IPC for the set of branch predictors running the selected benchmark programs for a 16KB-256KB instruction cache sizes. Some of benchmarks--*compress*, *m88ksim*-- fit even in an 4K-8K instruction cache. However, for others, Fig. 17 shows that instruction cache size affects performance. With the PAp, hybrid, and bimodal predictors increasing cache size from 4K to 16K improves performance by 30 to 50 percent. Increasing cache size from 8K to 128K nearly doubles performance for *gcc*, *go*, *perl* and *vortex*. These results show that the instruction cache miss reduction, simply by increasing the cache size, more than compensates for loss of branch prediction accuracy.

F. Effects of History Register Length

Fig. 18 shows the misprediction rate as a function of the history register length. The misprediction rate of the PAp and gshare predictors initially falls steadily before flatting out at a misprediction rate of around 8%. In general, however, there is a little benefit from increasing the BHR length beyond 16-bits for most of the predictor

schemes. Hybrid predictor is the less benefit scheme, while gshare is remarkably benefit from increasing BHR. As history register lengths increase, predictors require an increasing number of counter initializations and therefore suffer an increasing numbers of initial mispredictions.

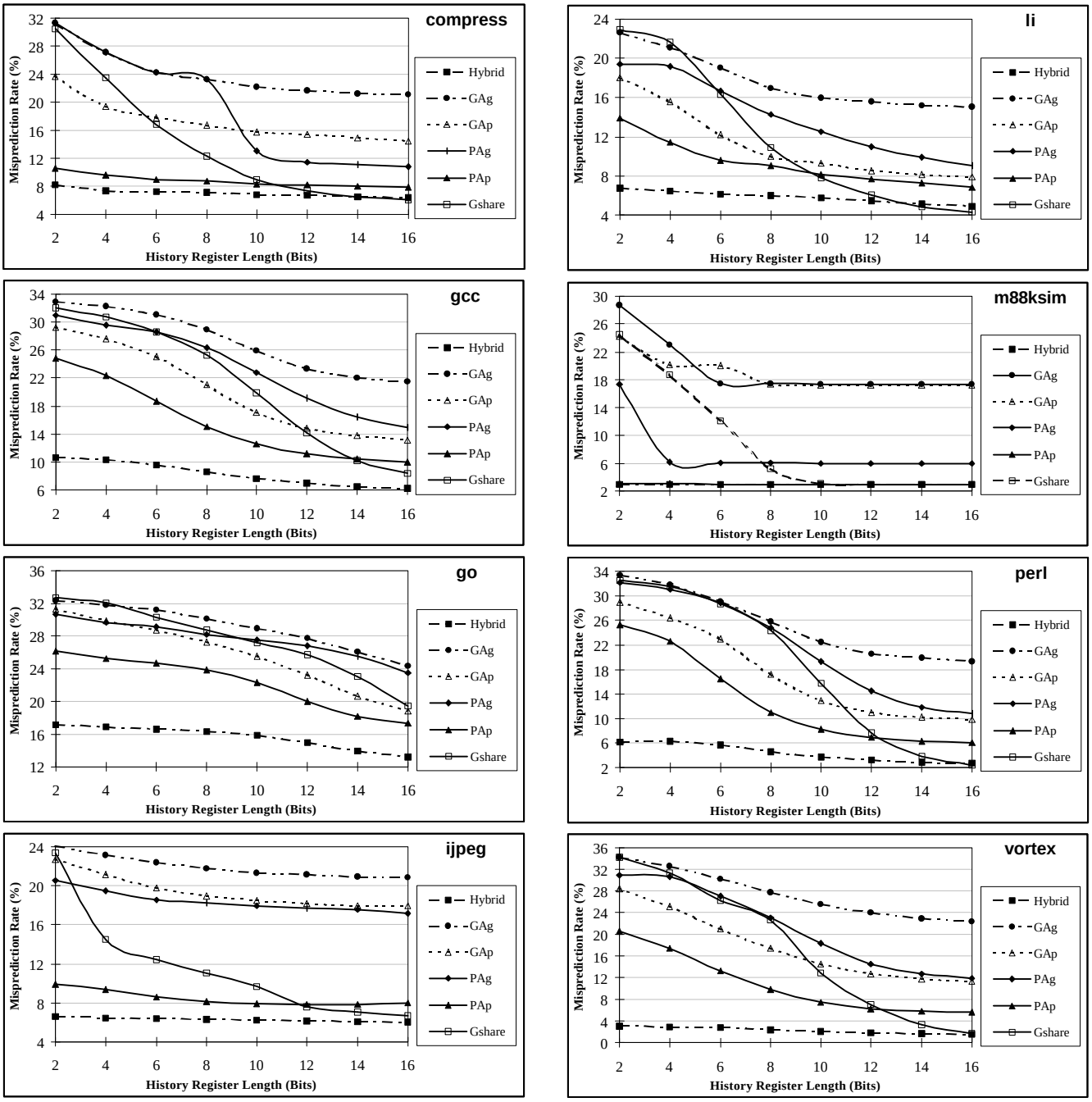


Fig. 18 A Comparison of Predictor Misprediction Rate as a Function of History Register Length (RUU = 256, Instruction Cache = 64K)

VII. CONCLUSIONS

Supporting the conditional execution of instructions is a technique that can have a significant impact on the performance of most ILP processors. We surveyed the most widely used branch prediction techniques in an out-of-order superscalar processor and evaluate it using the SPECint95 benchmarks. The performance impact of branch misprediction, the instruction-window size, machine width, the predictor size, first-level instruction-cache size, and history register length have been investigated.

The results show that predictors which do not use global history registers (bimodal and PAx), or which hash the global history register with the branch address or other values (gshare) will benefit from the predictor table interference reduction and, to some extend, increasing the BHR lengths. PAp and gshare is outperforms most of the individual predictors examined. Hybrid is the most promising predictor scheme available.

We have shown that increasing branch prediction accuracy does not necessarily mean higher processor performance. For example, better instruction cache performance may decrease prediction accuracy, but the reduced distance

between branch mispredictions is compensated by a lower cache miss rate, and a higher fetch width, which increase the speed at which instructions are provided.

REFERENCES

[1] D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architecture: A Design Space Approach*, Addison-Wesley, 1997.

[2] Shen draft book, or Shen Lecture-note

[3] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Aug. 1995*.

[4] G. P. Jones and N. P. Topham, "A comparison of superscalar and decoupled access/execute architecture," *Proc. of 26th Annual International Symposium on Microarchitecture*, Dec. 1993, pp.100-103.

[5] G.S. Sohi and A. Roth, "Speculative Multithreaded Processors," *Computer, Vol.34, No.4, 2001*, pp.66-73.

[6] E. Rotenberg, S. Bennett, and J.E. Smith, "Trace Cache A Low-Latency Approach to High-Bandwidth Instruction Fetch," *Proc. Int'l Symp. Microarchitecture*, IEEE CS Press, Los Alamitos, Calif., 1996, pp.24-34.

[7] S. Vajapeyam and M. Mitra, "Improving Superscalar Instruction Dispatch and Issue by exploiting Dynamic Code Sequences," *Proc. Int'l Symp. Computer Architecture*, ACM Press, New York, 1997, p.12.

[8] J.E. Smith, "Instruction-Level Distributed Processing," *Computer, Vol.34, No.4, 2001*, pp.59-65.

[9] A. Uht and V. Sindagi, "Disjoint Eager Execution: An Optimal form of Speculative Execution," Proc. 28th Int'l Symp. Microarchitecture, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 313-325.

[10] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark, "Multipath execution: Opportunities and limits," Proc. Intl. Conf. On Supercomputing, July 1998.

[11] Po-Yung Chang, et. al., "Using Predicated Execution to Improve the performance of Dynamically Scheduled Machine with Speculative Execution,"

[12] S. Mahlke, R. Hank, J. McCormick, D. August, and W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," Proc. of 22nd Intl. Symp. on Computer Architecture, June 1995.

[13] G. Tyson and M. Farrens, "Evaluating the Effects of Predicated Execution on Branch Prediction,"

[14] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of Control Independence in Superscalar Processors,"

[15] C. Young and M. Smith, " Static Correlated Branch Prediction," *ACM Transaction on Programming Languages and Systems*, May 1999.

[16] J.E. Smith, "A study of Branch Prediction Strategies," *Proc. of the 8th Annual International Symposium on Computer Architecture*, pp.135-148, May 1981.

[17] T. Ball and J. Larus, "Branch prediction for free," *Proc. of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 300-313, June 1993.

[18] B. Calder, D. Grunwald, and D. Lindsay, "Corpus-based static branch prediction," *Proc. ACM SIGPLAN Conf. on Programming language design and implementation*, pp.79-92, 1995.

[19] S. Pan, K. So, and J. Rahmeh, " Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *5th International Conference on Architecture Support for Programming Languages and Operating Systems*, Oct. 1992.

[20] T. -Y. Yeh and Y.N. Patt, "Two-level Adaptive Training Branch Prediction" *24th ACM/IEEE International Symposium on Microarchitecture*, Nov 1991.

[21] T. -Y. Yeh and Y.N. Patt, "Alternative Implementation of Two-Level Adaptive Branch Prediction," *19th Annual International Symposium on Computer Architecture*, pp.124-134, May 1992.

[22] C. Young, N. Gloy and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. 22nd International Symposium on Computer Architecture*, June 1995.

[23] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer, Vol. 17, No.1*, Jan. 1984.

[24] S. McFarling, "Combining Branch Predictors," WRL Technical Note TN-36, Digital Equipment Corporation, June 1993.

[25] R. Nair, "Dynamic path-based branch correlation," Proc. of the 28th Annual International Symposium on Microarchitecture, 1995.

[26] S. Sechrest, C-C Lee and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors," *Proc 23rd International Symposium on Computer Architecture*, May 1996.

[27] C.-C. Lee C. Chen and T. Mudge, "The bi-mode branch predictor," Proc. Of the 30th Annual International Symposium on Microarchitecture, November 1997.

[28] A. Eden and T. Mudge, "The YAGS branch prediction scheme," Proc. Of the 31st *Annual ACM/IEEE International Symposium on Microarchitecture*, November 1998.

[29] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, "The Agree Predictor: A mechanism for reducing negative branch history interference," Proc. Of the 24th International Symposium on Computer Architecture, June 1997.

[30] T.-Y. Yeh, D.T. Marr, and Y.N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," Proc. of the 7th ACM Conference on Supercomputing, pp.67-76, July 1993.

[31] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block a head branch predictors," Proc. of the 7th Intl. Conf. On Architectural Support for Programming Languages and Operating Systems, pp. 116-127, Oct. 1996.

[32] S. Onder, J. Xu, and R. Gupta, "Caching and predicting branch sequences for improved fetch effectiveness," Proc. Intl Conf. On Parallel Architectures and Compilation Techniques, Oct. 1999.

[33] K. Driesen and U. Hölze, "The cascaded predictor: Economical and adaptive branch target prediction," Proc. of the 31th Intl. Symp. on microarchitecture, Dec. 1998.

[34] R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro, 19(2):24-36, March/April 1999*.

[35] P.-Y. Chang, E. Hao, and Y. N. Patt, "Alternative implementations of hybrid branch predictors," *Proc. of the 28th Annual International Symposium on Computer Microarchitecture*, 1995, pp. 252-257.

[36] K. Driesen and U Holzle, " Accurate indirect branch prediction," *Proc. of the 25th Annual International Symposium on Computer Architecture*, pp.167-178, 1998.

[37] T. Juan, S. Sanjeevan, and J. J. Navarro, "Dynamic History-length fitting: A third level of adaptivity for branch prediction," Proc. of the 25th Annual International Symposium on Computer Architecture, pp. 155-166, 1998.

[38] P.-Y. Chang, E. Hao, and Y. N. Patt, "Predicting indirect jumps using a target cache," *Proc. of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 274-283.

[39] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus IPC," *Proc. Of the 27th Annual International symposium on Computer Architecture*, pp.248-259, June 2000.

[40] D. Jimenez, S. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," *Proc. Of the 33rd Annual International Symposium on Microarchitecture*, December 2000.

[41] D. C. Burger and T. M. Austin, "The Simplescalar Tool Set, Version 2.0," *Computer Architecture News*, 25 (3), pp. 13-25, June 1997.

[42] D. C. Burger, T.M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set'" Technical Report TR-1308, Computer Science Dept., Univ. of Wisconsin-Madison, July 1996.

[43] K. Skadron and P. S. Ahuja, "HydraScalar: A Multipath-Capable Simulator," IEEE Computer Society Technical Committee on Computer Architecture Newsletter, January 2001, pp.65-70.

[44] G. Hinton, et al., "The Microarchitecture of the Pentium® 4 Processor," *Intel Technology Journal Q1*, 2001.

[45] "Microprocessors of the 21st Century," IEEE Micro, Vol. 20, No.4-6, 2000.

[46] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," Proc. ISCA-24, pp. 292-303, June 1997.

[47] E. Sprangle, R.S. Chappell, M. Alsup, and Y.N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," Proc. ISCA-24, pp.284-91, June 1997.

[48] M. Evers, P.-Y. Chang, and Y.N. Patt, "Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches," Proc. ISCA-23, pp.3-11, May 1996.

[49] D. Grunwald, D. Lindsay, and B. Zorn, "Static methods in hybrid branch prediction," Proc. PACT, Oct. 1998.

[50] J. Emer and N. Gloy, "A Language for describing Predictors and its Application to Automatic Synthesis," *Proc. 24th International Symposium on Computer Architecture*, Jun. 1997.

[51] S-T Pan, K. So and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1992, pp.76-84.

[52] Digital Semiconductor, *Alpha 21164 Microprocessor*: Hardware Reference Manual, Apr. 1995.

[53] K. C. Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, 16(2):28-40, Apr. 1996.

[54] L. Gwennap, "Intel's P6 uses decoupled superscalar design," Microprocessor Report, pp.9-15, Feb.16,1995.

[55] L. Gwennap, "Digital 21264 sets new standard," Microprocessor Report, pp.11-16, Oct. 28,1996.

[56] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," Proc. of the 24th Int'l Conf. On Computer Architecture, 1997.

[57] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkum, “Confidence Estimation for speculation control, “ Proc. of the 25th Annual Int’l Symp. on Computer Architecture, Barcelona, 1998, pp.122-13.
[58] C. Price, MIPS IV Instruction Set, Revision 3.1, MIPS Technologies, Inc., Mountain View, Calif., Jan. 1995.
[59] T.-Y. Yeh and Y.N. Patt, “A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History,” Proc. 20th Ann. Int’l Symp. Computer Architecture, pp.257-266, May 1993.
[60] “Standard Performance Evaluation Corporation (SPEC)” http://www.specbench.org/
[61] C. Navarro, A. Ramirez, J. L. Larriba-Pey, and M. Valero, “On the Performance of Fetch Engines running dss Workloads,” Proc. of the Intl. Euro-Par Conference, Aug. 2000.
[62] K. Skadron et al., “Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques,” IEEE Transactions on Computers, Vol. 48, No. 11, November 1999, pp.1260-1281.
[63] P-Y. Chang, M. Evers, and Y. Patt, “Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference,” *Proc. of the Parallel Architecture and Compilation Techniques*, 1996.