Predicting Indirect Jumps In Systems Using a Branch Address Cache Multiple Branch Predictor

David M. Koppelman
Department of Electrical & Computer Engineering, Louisiana State University
koppel@ee.lsu.edu

Abstract

Ordinary branch predictors are not adequate for wide-issue processors since they cannot predict ahead more than one basic block per cycle, usually fewer instructions than a wide-issue processor can handle. Within a cycle a multiple branch predictor (MBP) *provides the information necessary for a processor to fetch instructions on a predicted execution path even if the path contains multiple taken branches, an improvement over classical branch predictors that overcomes the bottleneck encountered by wide issue machines. The MBP makes use of a PC-indexed* branch address cache *(BAC) to store basic block information. The MBP of Yeh, Marr, and Patt, and related mechanisms predict only branches, no special provisions are made for predicting indirect jumps which make up a significant portion of control transfers in some popular programs. A strategy for predicting indirect jumps in a system using a MBP is described and analyzed. In it the BAC, in addition to its usual role, is used as a branch-history-indexed jump target buffer. This dual use exploits BAC size, history-induced sparsity, and the fact that certain entries in the BAC can be shared with little impact on performance. Jump prediction is provided at almost no cost. Compared to a 16-entry GHR-indexed JTB, the scheme realizes significant performance improvement on some benchmarks, for example 15% speedup over a system using a small jump target buffer.*

1. Introduction

One bottleneck to superscalar processor performance is fetch efficiency. Even on 8-way processors using ordinary branch predictors and caches performance suffers considerably if the instructions decoded each cycle may not span a basic block. To overcome this limit several advanced fetch mechanisms have been proposed, including the multiple branch predictor (MBP) of Yeh, Marr, and Patt [16]. At each cycle, using an instruction address and global branch outcome history (the contents of a *global history register* [GHR]), an *order d* multiple branch predictor predicts the next

d basic blocks. The prediction is provided to a multiported cache and instruction reorder hardware which prepares the retrieved instructions for decode. The prediction is made using data from a conventional, though multi-ported, pattern history table and a *branch address cache (BAC)*. Each entry in the PC-indexed BAC stores information about basic blocks within $d-1$ control transfers (branches, jumps, etc.) of the current address, forming a *block tree*. The prediction is made by using a branch predictor to find a path through the block tree [16].

Yeh *et al* show that the MBP is quite effective at eliminating the fetch bottleneck, increasing the pressure on other parts of the system, including branch prediction, cache performance, and inherent ILP bottlenecks.

One possible objection to a multiple branch predictor is the size and utilization of the branch address cache. Each entry in an order d BAC entry has $2^d - 1$ nodes, many of which will rarely be used since most branches are highly biased. In simulations reported here, an order-2 MBP provided the best or close to the best performance when order was varied while holding BAC size constant, so the size is not prohibitive.

In the work reported here BAC utilization is improved by using parts of some BAC entries to store indirect jump targets. Thus the BAC serves double-duty as a GHR-indexed *jump target buffer (JTB)*, used only for non-return indirect jumps. This sharing is effective because a GHR-indexed JTB is used only sparsely (affecting few entries) and because the borrowing of a leaf only reduces the number of instructions that can be predicted on the affected path. In this way nearly half of the BAC capacity can be used as a JTB.

Indirect jump prediction accuracy is strongly affected by JTB size [2]; by sharing storage with the BAC the benefit of a large JTB can be had essentially for free.

Whether a BAC can support this sharing without compromising its fetch capability was evaluated by simulation of an aggressive though realistic 8-way system using an order 2 BAC using the shared JTB, and for comparison several other indirect jumps prediction methods. In addition to having the BAC serve as a JTB, indirect jumps are predicted by treating a jump target as a branch target and updating the block node on a misprediction, this is similar to a JTB using a PC and up to $d-1$ bits of branch history to form the index. Conventional PC- and GHR-indexed JTBs were also evaluated.

The remainder of this paper is organized as follows. In the following section details on the

MBP as analyzed here are presented. Indirect jump strategies appear in Section 3. Simulator and benchmark details are provided in Section 4. Results of experiments showing the optimal MBP order are in Section 5. Other experimental results and discussion appear in Section 6 followed by a discussion of prior work in Section 7. Conclusions follow.

2. Multiple Branch Prediction

2.1. Overview

The shared JTB/BAC system to be evaluated is based on the multiple branch predictor of Yeh, Marr, and Patt [16] though with predictor/instruction cache decoupling similar to the one described by Reinman, Calder, and Austin[12,13]. The major components of this system are a d-order MBP which predicts up to d basic blocks per cycle, a multi-ported instruction cache, and a mechanism to construct cache requests and *decode groups* using the predicted blocks. (A decode group is a set of instructions to be offered to the processor in a single cycle for decoding.)

The d-order MBP predicts up to d basic blocks per cycle, there are no alignment or small size restrictions on the predicted blocks. For each predicted block a *predicted path entry (PPE)* is constructed and placed in a *predicted path queue (PPQ)*. A PPE consists of the address of the first instruction in the block, the length of the block, and information needed for recovery.

PPEs are removed from the PPQ and are used to construct an instruction cache fetch request, which is sent to the cache. Data arriving from the cache is arranged into a *decode group* and enqueued in a *decode queue*. The processor dequeues up to one decode group per cycle. A PPE can span any number of decode groups, the number of PPEs "in" a decode group is limited by the number of instruction cache ports.

A decode group contains the instructions to be decoded, a PPE index used for recovery, and a bit indicating whether the PPE is *complete*. A PPE which is not complete describes a basic block of unknown length; the processor signals the MBP when the terminating control-transfer instruction (CTI) is encountered.

On a misprediction the processor provides the correct target address and PPE index; the index is used to recover the GHR and other information from the queue of consumed PPEs.

In the description given by Yeh *et al* predictions are made only when the instruction cache is ready. In a decoupled system, such as the one described above, predictions are enqueued allowing

Table 1. Branch Address Cache Entry Fields

Name, Size	Description
Tag, 17	BAC entry tag (part of address not used to index the BAC). The BAC is indexed using the address of the first instruction in the block at the root of the block tree.
Valid, 1	Set if entry valid.
Block Tree, $44(2^d - 1)$	Tree describing basic blocks within $d - 1$ control transfers of the root. Space allocated for complete binary tree, $2^d - 1$ nodes.

Table 2. Block Tree Node Fields

Name, Size	Description
Type, 4	Type of control transfer at end of block, whether node is shared, or other state of node. Five bits needed to distinguish minor CTI variations, such as annulled branches.
Length, 10	Number of instructions in basic block.
Target, 30	Target address of control transfer at end of block.

the predictor to run ahead of the icache. This allows fetch to occur at full speed even though in some cycles fewer than d basic blocks are predicted. A system with decoupling can attain higher performance or may use a lower-order predictor. (Reinman *et al* show how decoupling can be used for icache prefetch [13], this capability is not exploited here.)

Decoupling is important for the operation of the shared BAC/JTB for two reasons. First, a shared block tree leaf on the predicted path reduces the size of the prediction. Second, a prediction cycle is lost when the BAC is used as a JTB.

2.2. Multiple Branch Prediction

A d-order multiple branch predictor is designed to sustain a predict rate of d basic blocks per cycle. The predictor used is similar to the one described by Yeh *et al* [16]. The predictor consists of two tables, a *pattern history table (PHT)* and a *branch address cache (BAC)*.

The PHT is used, in effect, like the PHT in a GAg branch predictor [17] to predict d branches in a cycle. In such a predictor the PHT, a 2^n-entry table, is typically indexed using the outcome of the last n branches. Each entry is a two bit saturating counter used to predict the branch. To sustain d predictions per cycle the PHT in a MBP is d-ported and provides two-bit counters for all

$2^d - 1$ branches that might be encountered given only the GHR state at the beginning of the cycle. See [16] for details.

Because branch addresses are not available at the beginning of a cycle schemes such as gshare [8] are not possible. (The address of the first block could be used, but that would increase warm-up time because the same branch can appear in several BAC entries.) To compensate the system analyzed here shifts in four bits of the target address for indirect jumps, avoiding the disorientation that would occur after returns and other indirect jumps.

The BAC is indexed using the program counter (PC), which usually points to the first instruction in a basic block in this system. A BAC entry consists of a tag, state, and a *block tree* describing basic blocks within d control transfers of the PC, see Table 1 for a list of fields and typical sizes. Each block tree node consists of a CTI field, a length, and a target address; these fields are listed in Table 2. The CTI field indicates the type of control transfer terminating the block (branch, jump, etc) or whether the block is unused or being borrowed for a JTB entry. The length indicates the size of the block, and the target indicates the address on the taken path. Because an entire tree is stored, only low-order MBPs are feasible.

Tree leaves can describe any type of CTI, internal nodes are restricted to instructions with known target addresses (though the direction may be uncertain). For the systems simulated jump and link instructions, used for procedure returns and other indirect jumps, must be leaves. (Procedure calls on the simulated system use displacement addressing.)

2.3. MBP Operation

At the beginning of the cycle a BAC entry and PHT counters are retrieved. If the tag matches, blocks on the predicted path are selected (based on the predictions) and the addresses of those blocks are computed, forming predicted path entries. Also generated by the end of the cycle is a next PC and a next GHR. Note that for a order 2 MBP, the type considered, two of three nodes are frequently used per lookup. If the last node on the predicted path looks like a return the next PC is predicted using a return-address stack. The handling of indirect jumps is covered in the next section.

If the tag does not match an incomplete PPE is formed, consisting of the correct start address (the PC) a maximum length and an *unreached* type for the CTI. The retrieved BAC will be

replaced with a new one, filled while the processor uses ordinary fetching.

A disadvantage of the MBP schemes over simpler fetch mechanisms is the time needed to redirect fetch after a misprediction, or for some other reason. In addition to the cache lookup time a cycle is needed to arrange instructions in decode order. It is possible for the instruction cache to start a fetch in the cycle the processor initiates a redirect, however in the systems simulated here the redirect address is given to the MBP, delaying an extra cycle.

3. Indirect Jump Prediction

Indirect jumps make up a significant portion of some programs, for example, perl, TEX, and gcc and the cycles lost to either not predicting them or mispredicting them can rival the cycles lost to mispredicting branches.

3.1. Existing Jump Prediction Schemes

Existing systems use return-address stacks to predict what look like procedure returns and branch target buffers to predict other indirect jumps and branches [7,11]. Advanced PC- and GHR-indexed jump target buffers have been described in the literature [2]. The target predicted using a PC-indexed JTB is the last target that the jump computed. The target predicted using a GHR-indexed JTB is the last target the jump used with the same GHR value. (Possible the same GHR used by the branch predictor.) PC-indexed predictors make more efficient use of space but large GHR-indexed predictors have higher accuracies. Since an entire target address must be stored, GHR-indexed JTBs are 16 to 32 times larger than PHT using the same history size.

3.2. Node-Stored Jump Targets

In a system using an MBP, the BAC is well suited to serve as a JTB. Two methods will be analyzed. In the first, called *Node-Stored*, the block tree node for an indirect jump stores its own predicted target, which is the last target generated when that node was used. The target is updated after each misprediction. This simple system may well have been used by others without note; here its performance is analyzed.

3.3. Leaf-Stored Jump Targets

In the second, called *Leaf-Shared*, the target of an indirect jump is predicted in the cycle after it

is encountered. The BAC is indexed using part of the GHR (rather than the PC) and another part is used to select one of the leaf nodes. (Borrowing leaf nodes allows the shared entry to fetch more blocks than if an internal node were borrowed.) If the CTI type for the retrieved node is marked shared its target is used, otherwise it is marked shared and fetch stalls until the jump resolves. When the jump resolves the node will be updated unless a correct prediction was made. This may overwrite a node in use, if so one fewer block is predicted on that path. In the systems simulated a leaf remains shared until a new path reaches that leaf, usually after the BAC entry is replaced.

3.4. Contrasts

In terms of prediction method the Leaf-Shared scheme is similar to the GHR-Indexed JTB scheme, the only difference is that something other than another jump can replace a "JTB" entry in the Leaf-Shared scheme. The added hardware cost of the shared scheme is small, given the MBP hardware already present. The predicted target is produced right where it's needed and the multiplexors needed to select the correct tree node are already present. The timing of the two schemes are similar: in both cases the next cycle is used to predict the jump outcome. There is the potential negative performance impact of overwriting too many tree nodes. That was tested in the simulations by using a MBP order that gave optimal results using an ordinary GHR-Indexed jump predictor.

The Node-Stored scheme is similar to the PC-Indexed JTB, with two differences. If a jump instruction is at more than one node (in the same or different trees), each jump can have its own target, so there is some correlation with branch history, albeit a short history. The second difference is that in the Node-Stored scheme the prediction is produced in the cycle it is encountered, rather than one cycle later.

The significance of these differences will be evaluated in the experiments.

4. Evaluation

4.1. Simulator

The systems were analyzed using RSIM [9], a detailed microarchitecture simulator. Modifications were made to simulate multiple branch prediction and the sharing scheme described here, other unrelated modifications were made; that is, they impact the reported performance of systems.

RSIM is a microarchitecture simulator which simulates a dynamically scheduled superscalar processor and memory system. The processor implements the SPARC V8 ISA [14]. Benchmark programs are statically linked and use a special startup file, but other aspects of their preparation are unmodified, including the use of standard system libraries. System calls are not simulated, though their impact on the execution of the chosen benchmarks should be small.

Dynamic execution is aggressive: The register map used for renaming is checkpointed when branches or jumps are decoded so that recovery can start when mispredicted instructions resolve. Exception recovery is initiated when the faulting instruction is ready to commit.

4.2. Benchmark Programs

The simulated programs come from the SPEC suites, though using reduced input sizes to reduce simulation time. The four programs used are bzip2, gcc (cc1), perl, and TEX. Benchmark bzip2 is used to compress a copy of the GNU General Public License; gcc is used to compile (with O3 optimization) a program for finding hidden words in text, perl runs a script that analyzes a Web server log, and TEX is run on the GNU Emacs quick reference card. The number of committed instructions ranges from about 23 million for bzip2 to 118 million for gcc.

4.3. Configurations

The simulated system was sized to represent a system realizable in the next ten years. The issue width was limited to 8 by branch prediction accuracy; in wider systems the additional instructions fetched were mostly squash fodder. Note that improved branch predictors such as a hybrid local/ global predictor [8] are difficult to implement on an MBP since block addresses (other than the first) are not known at the time that table lookup should start. Prediction techniques used in next-trace predictors, which share the block knowledge problem, such as a shorter warm-up history and restoring history on a procedure return [5] could have been used except the cost of the BAC along with a next trace predictor would be hard to justify.

A large instruction cache was chosen to focus attention on the fetch mechanism itself; the icache hit ratio is near 100%. The level-1 and level-2 data caches are 2 KiB (2^{10} B) and 32 KiB, respectively. Though they may seem small, for the problem sizes used they produce hit ratios of about 70% for level 1 and over 95% for level 2. There are an ample number of functional units and a large reorder

Table 3. Base Configuration Parameters

Simulation Parameter	Value
Issue	8-way Superscalar
Reorder Buffer	512 instructions
Return-Address Stack	8 entries
ICache	786,432 B
ICache	3 way, 256-B Line
L1 Data Cache	4-way,32-B Line, 2 KiB
L1 Hit Latency	1 cycle
L2 Data Cache	8-way,32-B Line, 32 KiB
L2 Hit Latency	7 cycles
L2 Miss Latency	≈ 40 cycles
Integer FU	8
Floating-Point FU	4
Address Units	4

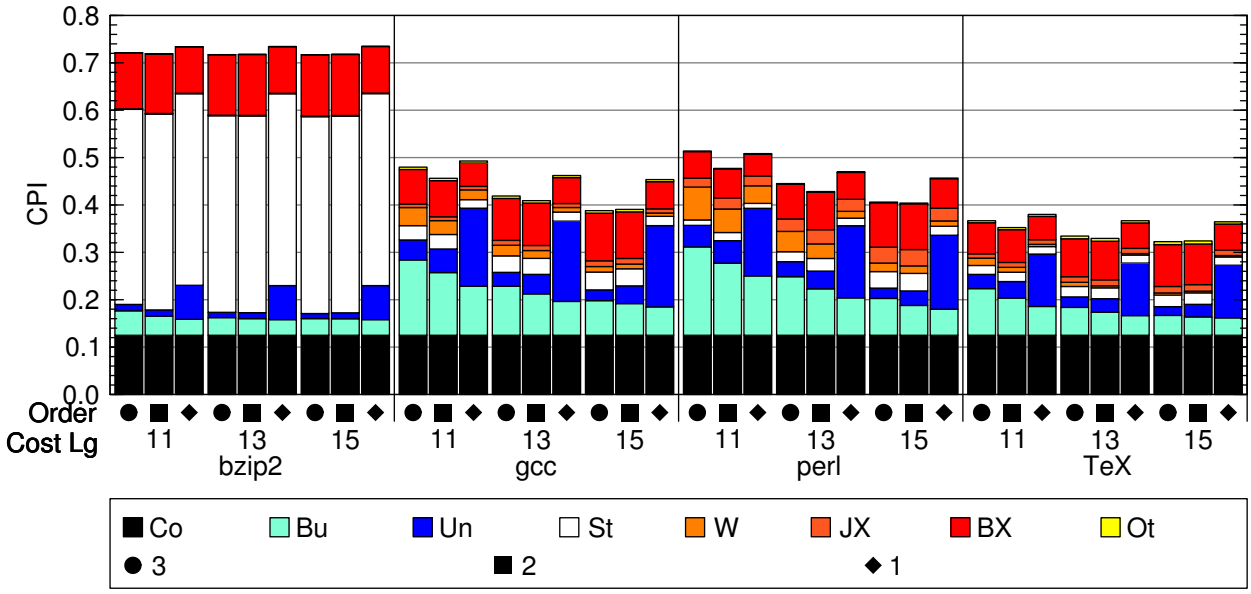

Figure 1. Average instruction execution time (CPI) for three groups of equal-cost MBP with varying orders (tree sizes) using ordinary GHR-Indexed jump prediction.

buffer, see the table below.

5. Optimal MBP Order

The MBP order was chosen to maximize performance. Using the parameters given above, systems were simulated in which the order was varied but the capacity of the BAC held constant. The capacity of an order d MBP with a 2^b-entry BAC is proportional to 2^{d+b}, assuming that a block tree node is the same size as the non-tree information in a BAC entry.

Figure 1 shows the CPI for equal-cost order 1, 2, and 3 systems at three different costs. For the largest systems an order 3 predictor is fastest, but only marginally; for the others an order 2 is

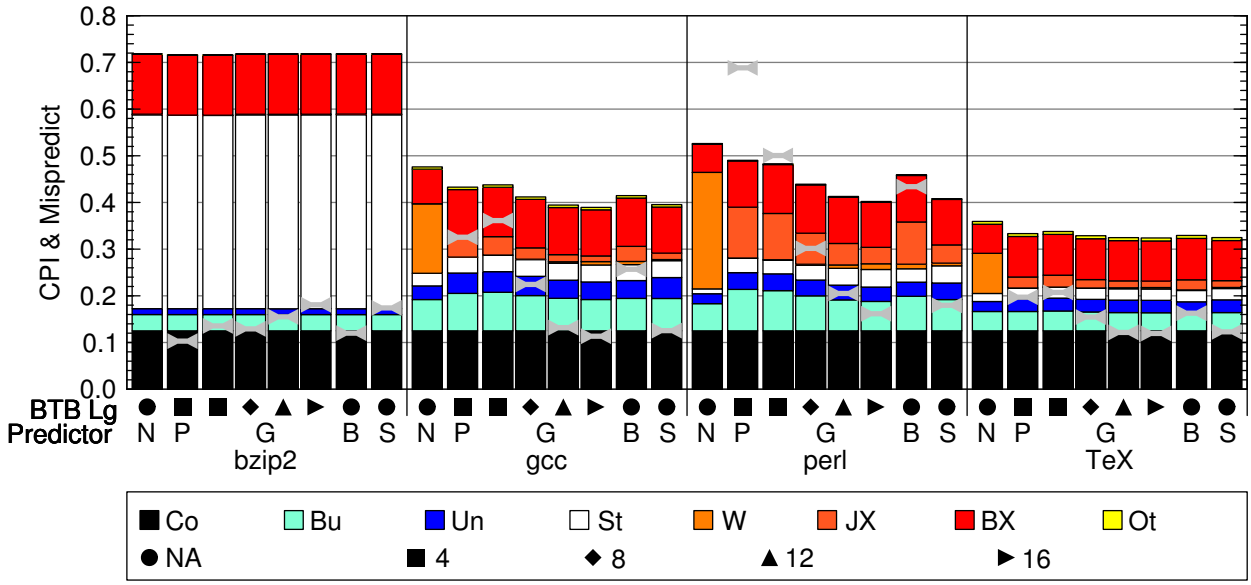

Figure 2. Performance of jump predictors on order 2, 2^{13}-Entry BAC MBP systems. Bowties show jump mis-prediction rate.

fastest, and so that is the size used.

The segments show contributers to CPI by categorizing instructions that were decoded or reasons why instructions weren't. Segments marked **Co** show committed instructions, their height is the ideal CPI. The effectiveness of the fetch mechanism is shown by segments marked **Un** , which show the impact of unfilled slots in a partially filled decode group. This accounts for a good fraction of the execution time in first-order MBPs. The other segments are described in Section 6.

6. Experiments

6.1. Performance

The effectiveness of Leaf-Shared and Node-Stored jump prediction were evaluated by comparison with systems using an identical MBP but using either no jump prediction, PC-Indexed jump prediction or GHR-Indexed jump prediction at a range of table sizes.

In each set of simulations the BAC is fixed, so the PC-Indexed and GHR-Indexed predictors use more total storage. It would be difficult to make the total cost of each system identical without resorting to non-power-of-2 table sizes or oddly proportioned tables. The base system used for comparison uses a 16-entry GHR-Indexed JTB, larger JTBs are also simulated for comparison purposes. The systems use an order 2 MBP, the one having the highest performance for a fixed amount of BAC storage.

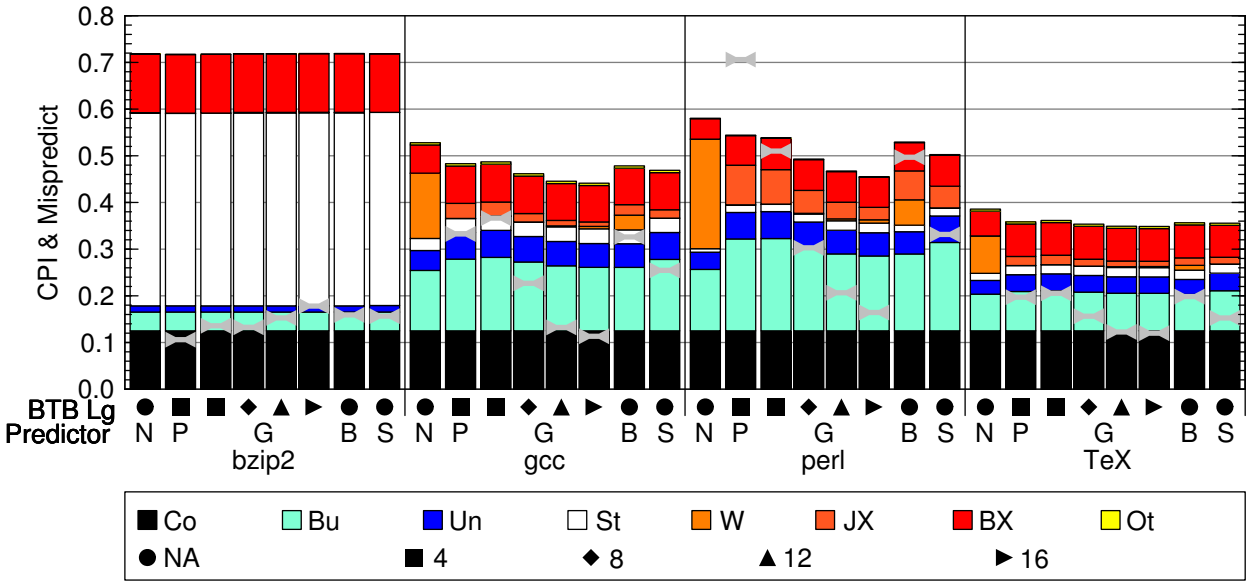

Figure 3. Performance of jump predictors on order 2, 2^9-Entry BAC MBP systems. Bowties show jump mis-prediction rate.

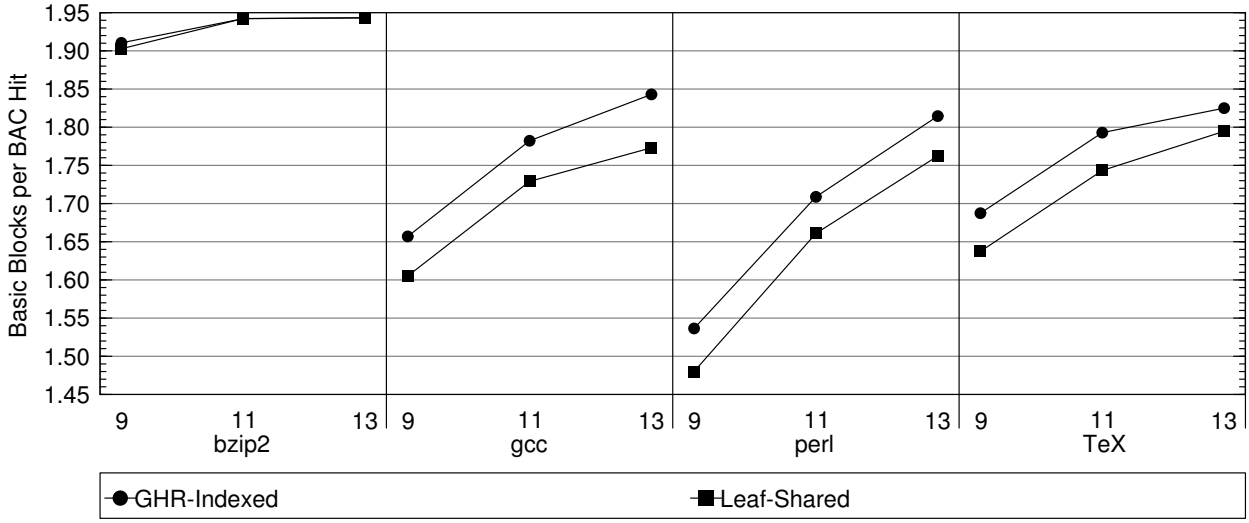

Figure 4. Number of basic blocks per BAC hit versus BAC size, x-axis labeled with $\log_2$ of number of BAC entries.

Average instruction execution time (CPI, cycles per instruction) for the various schemes is plotted in Figure 2 for systems using a 8192-entry BAC. At this size the Leaf-Shared scheme uses one quarter the number of entries of the largest GHR-Indexed JBT predictor tested. The first bar in each group is for a system without jump prediction, labeled **N** , the second, **P** , uses a PC-indexed JBT; the next four bars are for GHR-Indexed predictors, **G** , and the last two are for Node-Stored, **B** , and Leaf-Shared, **S** , respectively.

Jump prediction of any kind has almost no impact on the first benchmark, for which there are few indirect jumps. For gcc and perl the benefit of jump prediction is large, while for TEX it is small.

The speedup obtained using a large GHR-Indexed JTB is comparable to that obtained using the Leaf-Shared scheme, about 10% for gcc and 15% for perl. The Node-Stored scheme achieves a modest speedup, its reduced prediction latency does not compensate for its lower prediction accuracy.

6.2. Performance Factors

The segments reveal how instruction fetch contributes to execution time by categorizing instructions that were fetched or why instructions were not fetched. The size of a segment is the number of instructions in a category divided by the total number of instructions committed.

The segments marked **Co** are for instructions that are fetched and committed. The segments marked **JX** show instructions that are fetched and then squashed due to a jump misprediction, the segments marked **BX** show instructions squashed due to branch mispredictions.

The size of the **JX** and **BX** are a function of the number of mispredictions and the number of cycles from decode to detection of the misprediction (resolution), so they are a good indicator of misprediction impact. For perl, which shows the greatest speedup, the impact of jump mispredictions is greater than branch mispredictions. The impact on gcc is more modest but still significant. There is a minor impact on TEX, and an insignificant one on bzip2.

The segment marked **W** shows the number of instructions that could have been fetched while waiting for an unpredicted jump to resolve. This is a major component of execution time in the systems without jump prediction. Though execution time is slowest in such systems fewer instructions are squashed which might translate to energy savings.

The segments marked **Un** show the number of empty slots in a partially filled fetch group, indicating the ability of the fetch mechanism to provide enough instructions. There is a minor difference between Shared and the GHR-Indexed with a 2^{16}-element JTB, due to shared nodes. However the number of empty slots is also high in systems with poor jump prediction because of the BAC created on mispredicted jumps.

The jump misprediction rate is shown with bowties in Figure 2, sharing the Y-Axis scale with

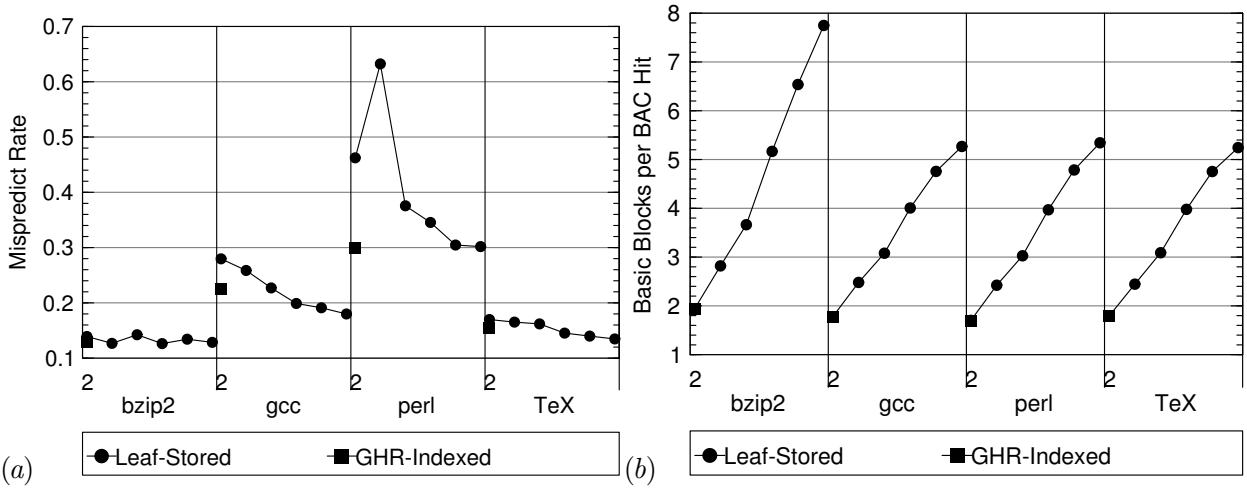

Figure 5. Jump mispredict rate and blocks per BAC Hit v. MBP order from 2 to 10. Graphs illustrate positive impact of large tree sizes (orders) on prediction accuracy.

CPI. Except for bzip2, the prediction accuracies are what one would expect from the CPIs. Accuracy running bzip2 suffers with larger GHR-Indexed predictors, probably because there are not enough jumps to warm up the JTB, shared or separate.

At 2^{13} entries the BAC provides the equivalent of a 2^{14}-entry JTB, smaller BAC perform less well because the shared JTB is smaller and because there is a greater chance of a shared node in an actively used block tree leaf. The degradation can be seen in Figure 3 which shows normalized execution time on a system using a 512-entry BAC. Speedups over a 16-entry JTB are more modest, about 4% for gcc, and 6% for perl.

Most of this degradation is due to the shorter global history, as can be inferred from the GHR-Indexed JTB data. Leaf borrowing might also degrade performance by reducing the number of occasions when the MBP can predict multiple blocks. This is not a major factor as can be seen in Figure 4 where the average number of basic blocks found per BAC hit is plotted versus BAC size for the Leaf-Shared and 2^{16}-entry GHR-Indexed predictors. There is a small but noticeable drop in the number of blocks, though on average the number of instructions is still larger than the fetch width.

6.3. MBP Order and Node-Stored Jump Prediction

One property of the Node-Stored predictor is that jump predictions are correlated with a small amount of branch history, approximately the branches from the root of the block tree to the jump,

but at a depth of two that history is insignificant. In Figure 5(a) the prediction accuracy of systems using a 2^8-entry GHR-Indexed JTB is compared to systems using a Node-Stored predictor for MBP systems from order 2 to the impractically high order 10. The prediction accuracy and performance (not shown) of the Node-Stored predictor at order 8 exceeds that of the GHR-Indexed predictor on an order 2 MBP. The average number of blocks used per BAC entry hit is plotted in Figure 5(b); the increase is nearly linear indicating that the trees are used to a substantial depth.

7. Related Work

7.1. Indirect Jump Prediction

An early method of predicting indirect jumps used a branch target buffer to store the target of an indirect jump; the JTB was used to store the targets of other control transfer instructions, and was primarily intended for branch prediction [7,11]. Such a classical JTB is indexed by instruction address and returns, among other things, a branch target. Designs more appropriate to systems using an instruction cache predict the line, and perhaps set on which a target will reside, reducing the number of address bits needed; for example those of Johnson [6] and Calder and Grunwald [1]. These schemes, used in some form in current processor designs, predict the same target for an indirect jump each time. (Though a separate return-address stack is used for return instructions.)

Chang, Hao, and Patt, show that the single-target "assumption" does not hold in popular benchmarks, and present and analyze several correlating jump target predictors [2]. They are all variations on a history-indexed JTB, which they call a *target cache*. They show significant improvement over designs that index using only the jump address.

7.2. Multiple Branch Predictors

The multiple branch predictor of Yeh, Marr, and Patt [16] was an early method of supplying instructions spanning multiple basic blocks to a superscalar processor. Their analysis focused on branch predictor performance and the prediction of indirect jumps was not considered. (Indirect jumps are handled by a single-block fetch mechanism in an earlier work [15].) The multiple branch predictor of Dutta and Franklin [4] also stores blocks reachable from an address, which they call a *tree-like subgraph*, but their predictor makes greater use of local history. Like Yeh *et al*, they do not discuss indirect branches. Neither Yeh *et al* nor Dutta and Franklin simulate equal-cost

systems to determine optimal tree depth.

Patel, Friendly, and Patt analyze several multiple branch predictors for a trace cache. The predictions are correlated with the start address of a trace: a PHT is indexed gshare-style, with the exclusive or of branch history and the trace start address. Several variations on what the PHT stores were analyzed, the best in the cost-equalized comparisons used the fewest counters, a counter for branches at distance 1, 2, 3, etc. (Rather than a tree of counters for all reachable blocks.) No mechanism is described for indirect jump prediction.

Conte, Menezes, Mills, and Patel [3] analyze various dual-port instruction cache configurations; they find the most flexible, using what they call a *collapsing buffer*, yields significant performance improvements. Such flexible multi-ported caches are assumed for the multiple branch prediction schemes discussed here.

Reinman, Calder, and Austin [12,13] describe an instruction fetch mechanism, the *Fetch Target Queue*, which is similar to an order 1 MBP, except that the single BAC node per entry stores information on several contiguous basic blocks, separated by highly not-taken biased branches. They show roughly 20% speedup over Yeh *et al Basic Block Target Buffer* [15] for small table sizes on 8-way systems, the speedup shrinks to a few percent when larger tables are used, indicating that the advantage is in compact table size. The comparisons of order 1 and higher MBP systems (see Figure 1) suggests that additional fetch capability is possible.

Like the MBP as implemented here, Reinman *et al*'s FTQ is decoupled, allowing the predictor to run ahead of the fetch mechanism. They use the queued predictions to prefetch the cache, whereas here decoupling is used to keep the fetch mechanism busy while the BAC is used for jump target lookups and when BAC entries are shortened due to sharing.

7.3. Path-Based Trace Predictors

The path-based next trace predictor of Jacobson, Rotenberg, and Smith is, in effect, a multiple branch and indirect jump predictor [5]. A *trace*, in this context, is a segment of the dynamic instruction sequence that has been encountered and cached for later use. Traces are identified by a *trace ID*, the address of the first instruction and the outcome of contained branches. (An indirect jump must be the last instruction in a trace.) Traces are predicted by using a hash of the last several trace IDs to index a table that returns a predicted trace ID. Thus, both branch directions and

indirect jump targets are being correlated with path history. The prediction accuracies attained, which also use variable history length and saved histories for calls, are very high [5].

Perhaps one disadvantage of this next-trace prediction mechanism is in the size of the history table needed. Each entry must store a trace ID, which must contain a substantial portion of an instruction address. A BAC also stores instruction addresses, however the number of times an address might appear in the table depends on the order, which is small, whereas in a next-trace predictor the number of places a trace ID appears depends on the number of encountered paths which lead to it, which can be large. Multiple branch predictors predict only branch direction, and so can use compact pattern history tables. A JTB can be added, as in the systems analyzed here, without having to shrink the number of entries in the PHT.

Trace caches can also use multiple branch predictors [10], and there are many differences between the two unrelated to prediction. For example, systems using trace caches can have simpler instruction caches and decode units.

8. Conclusions

Variations on indirect jumps prediction for a system using a decoupled multiple branch predictor were described and analyzed. The Leaf-Shared predictor takes advantage of the fact that some BAC leaves can be borrowed without a large impact on performance, the space is used to implement a large GHR-Indexed JTB, yielding higher performance on some benchmarks essentially for free. Also analyzed was the performance of a Node-Stored predictor, something which may have been implemented in earlier work but not specifically reported on.

The shared scheme is possible because some BAC entry nodes can be shared. BAC node sharing may be exploited for other purposes. For example, the data returned in a set-associative BAC lookup may be used to select one of two trees, or a single larger one. Preliminary work shows that an order 1 MBP with such *extended trees* performs as well as an order 2 MBP with the same BAC capacity. If they can be made to perform better they will be described in a future work.

9. References

[1] Brad Calder and Dirk Grunwald, "Next cache line and set prediction," in *Proceedings of the International Symposium on Computer Architecture*, June 1995, pp. 287–296.

[2] Po-Yung Chang, Eric Hao, and Yale N. Patt, "Target prediction for indirect jumps," in *Proceedings of the International Symposium on Computer Architecture*, June 1997, pp. 274–283.

[3] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the International Symposium on Computer Architecture,* June 1995, pp. 333–344.

[4] Simonjit Dutta and Manoj Franklin, "Control flow prediction with tree-like subgraphs for superscalar processors," International Symposium on Microarchitecture, December 1995, pp. 258–263.

[5] Quinn Jacobson, Eric Rotenberg, and James E. Smith, "Path-based next trace prediction," International Symposium on Microarchitecture, December 1997, pp.14–23.

[6] William Johnson, "Superscalar microprocessor design," Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

[7] Johnny K. F. Lee and Alan Jay Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer,* vol. 17, no. 1, pp. 6-22, January 1984.

[8] Scott McFarling, "Cache replacement with dynamic exclusion," in *Proceedings of the International Symposium on Computer Architecture,* May 1992, pp. 191–200.

[9] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve, "RSIM reference manual version 1.0," Rice University Dept. of Electrical and Computer Engineering, August 1997, Technical Report 9705.

[10] Sanjay Jeram Patel, Daniel Holmes Friendly, and Yale N. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Transactions on Computers,* vol. 48, no. 2, February 1999.

[11] Chris H. Perleberg and Alan Jay Smith, "Branch target buffer design and optimization," *IEEE Transactions on Computers,* vol. 42, no. 4, pp. 396-412, April 1993.

[12] Glenn Reinman, Todd Austin, and Brad Calder, "A scalable front-end architecture for fast instruction delivery," in *Proceedings of the International Symposium on Computer Architecture,* May 1999, pp. 234–245.

[13] Glenn Reinman, Brad Calder, and Todd Austin, "Optimizations enabled by a decoupled front-end architecture," *IEEE Transactions on Computers,* vol. 50, no. 4, pp. 338-355, April 2001.

[14] David L. Weaver and Tom Germond (eds.), "The SPARC architecture manual, Version 9," Englewood Cliffs, New Jersey: Prentice-Hall, 1994.

[15] Tse-Yu Yeh and Yale N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," International Symposium on Microarchitecture, December 1992, pp. 129–139.

[16] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proceedings of the International Conference on Supercomputing,* 1993, pp. 67–76.

[17] Tse-Yu Yeh and Yale N. Patt, "A comparison of dynamic branch predictors that use two levels of branch history," in *Proceedings of the International Symposium on Computer Architecture,* May 1993, pp. 257–266.