EXACT: Explicit Dynamic-Branch Prediction with Active Updates

Muawya Al-Otoom, Elliott Forbes, Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{mmaloto, jeforbe2, ericro}@ncsu.edu

ABSTRACT

Branches that depend directly or indirectly on load instructions are a leading cause of mispredictions by state-of-the-art branch predictors. For a branch of this type, there is a unique dynamic instance of the branch for each unique combination of producer-load addresses. Based on this definition, a study of mispredictions reveals two related problems:

(i) Global branch history often fails to distinguish between different dynamic branches. In this case, the predictor is unable to specialize predictions for different dynamic branches, causing mispredictions if their outcomes differ. Ideally, the remedy is to predict a dynamic branch using its program counter (PC) and the addresses of its producer loads, since this context uniquely identifies the dynamic branch. We call this context the identity, or ID, of the dynamic branch. In general, producer loads are unlikely to have generated their addresses when the dynamic branch is fetched. We show that the ID of a distant retired branch in the global branch stream combined with recent global branch history, is effective context for predicting the current branch.

(ii) Fixing the first problem exposes another problem. A store to an address on which a dynamic branch depends may flip its outcome when it is next encountered. With conventional passive updates, the branch suffers a misprediction before the predictor is retrained. We propose that stores to the memory addresses on which a dynamic branch depends, directly update its prediction in the predictor. This novel "active update" concept avoids mispredictions that are otherwise incurred by conventional passive training.

We highlight two practical features that enable large EXACT predictors: the prediction path is scalably pipelinable by virtue of its decoupled indexing strategy, and active updates are tolerant of 100s of cycles of latency making it ideal for virtualizing this component in the general-purpose memory hierarchy. We also present a compact form of the predictor that caches only dynamic instances of a static branch that differ from its overall bias.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles — *pipeline processors*

General Terms: Design, Performance.

CF'10, May 17–19, 2010, Bertinoro, Italy.

Keywords

branch prediction, superscalar processors, microarchitecture

1. INTRODUCTION

The important trend of placing multiple cores on a single chip has apparently shifted the research spotlight away from high-performance processor architectures and instruction-level parallelism, to chip-level architectures and thread-level parallelism. In reality, the diversity across and within workloads is too great to exclude either approach. Microprocessor companies continue to develop flagship high-performance cores (*e.g.*, AMD's K10 and Intel's Nehalem), even placing two, four, or more of these large cores on a single chip. Looking forward, a compelling strategy is to include a robust mix of core types in an Asymmetric Chip Multiprocessor (ACMP), *e.g.*, several flagship large cores and many simple cores, to support both low latency and high throughput [9][12][16][23]. Low latency is critical for serial workloads and serial regions of parallel workloads.

Continued microarchitecture performance scaling is hindered by many factors. One factor above all, the branch prediction bottleneck, constrains the ability to tackle other factors: the lookahead capability afforded by branch prediction exposes instruction-level parallelism (ILP) for combating data dependences and memory latency or acts as a catalyst for other speculative techniques aimed at extracting more ILP [5].

Today's best known branch predictors push the envelope of what is possible using global branch or path history as context for making predictions. While this context is basically the same used by precursor predictors since the advent of two-level adaptive branch prediction [25], clever combinations and organizations have yielded nearly perfect branch prediction on some programs and program phases. Yet, results in this paper show that branch history alone cannot scale accuracy in other programs beyond 90-95%. In these programs, the leading cause of branch mispredictions are branches that depend directly or indirectly on load instructions.

An example of this type of branch is depicted in Figure 1(a). It shows a static branch at program counter (PC) Z that depends on two static loads at PCs X and Y. At run-time, the static branch translates into many different dynamic branches corresponding to different combinations of load addresses. Two dynamic instances of the branch are shown in Figure 1(b). In the first instance, the two load instructions load from addresses A1 and B1, respectively. In the second instance, the two load instructions load from addresses A2 and B2, respectively. Thus, it is the combination of load addresses that distinguishes one dynamic branch from another. More generally, a dynamic branch is uniquely identified by the combination of its PC and the addresses of loads on which it

depends directly or indirectly. We call this combination the identity, or ID, of the dynamic branch. The IDs of the two dynamic branches in Figure 1(b) are {Z, A1, B1} and {Z, A2, B2}, respectively.

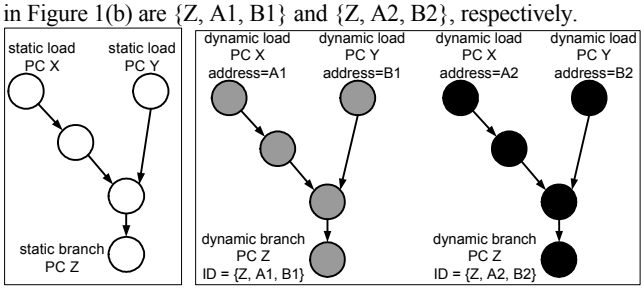


Figure 1. (a) A static branch that depends on two static loads. (b) Two dynamic instances of the static branch.

Many state-of-the-art branch predictors [10][11][13][15][18][20] exploit global branch history as context for predicting dynamic branches. When a static branch's PC is combined with global branch history, the static branch uses multiple prediction table entries instead of just one, an entry for each unique global branch history pattern preceding the branch. Ideally, this enables specializing predictions to different dynamic instances of the static branch. For example, for the static branch Z of Figure 1, a particular global branch history pattern, P1, may precede dynamic branch {Z, A1, B1} and a different pattern, P2, may precede dynamic branch {Z, A2, B2}. As shown in Figure 2(a), the two dynamic branches access different prediction table entries because they use different indices formed from {Z, P1} and {Z, P2}, respectively. This is advantageous if the two dynamic branches have different outcomes. They each have a dedicated entry in the prediction table for making different predictions. In a sense, the goal of combining PC with global branch history is to forecast which dynamic branch, *i.e.*, which ID, is currently being fetched and to provide a dedicated prediction for it.

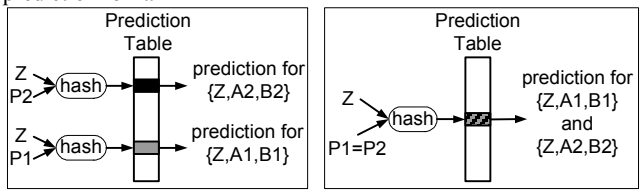


Figure 2. (a) Good scenario: different dynamic branches access different table entries (P1 ≠ P2). (b) Bad scenario: different dynamic branches access the same table entry (P1 = P2).

In Section 2, we diagnose the causes of mispredictions using the dynamic-branch framework defined above. We use very large versions of the *gselect* predictor [14][25] and *L-TAGE* predictor [18] (the latter predictor took first place in the most recent championship branch prediction [26]). The study reveals two problems:

1) *Insufficient specialization*. Often, global branch history, even very long history (640 bits) as used by some components in *L-TAGE*, does not distinguish between two or more dynamic branches. If these dynamic branches have different outcomes, some will be mispredicted because only a single prediction is available to predict all of them. This scenario is depicted in Figure 2(b) for the two dynamic branches in our running example. The problem is that the same branch history pattern precedes both dynamic branches (P1 = P2).

2) *Stores*. A store to an address on which a dynamic branch depends, may cause its outcome to be different the next time it is encountered. In this case, the dynamic branch will be mispredicted because its prediction table entry is stale with respect to the updated data in memory. The entry is only retrained after the misprediction is incurred.

In theory, the first problem can be addressed by using the dynamic branch's ID as a unique index into the prediction table. In general, however, producer loads are unlikely to have generated their addresses by the time the dynamic branch is fetched. We show that the ID of a distant retired branch in the global branch stream (*e.g.*, 20+ branches away) combined with recent global branch history, is effective context for predicting the current branch.

To address the second problem, we propose that stores to the addresses on which a dynamic branch depends, directly update its prediction in the predictor. This novel "active update" concept avoids mispredictions that are otherwise incurred by conventional passive training. With passive updates, the branch predictor is retrained after mispredicting. With active updates, the store updates both memory and the branch predictor, avoiding the misprediction by actively mirroring memory.

We call the proposed predictor EXACT, for "EXplicit dynamic-branch prediction with ACTive updates". "EX" conveys that dynamic branches are explicitly identified so that they can be provided dedicated predictions and "ACT" conveys that their predictions are actively updated by stores.

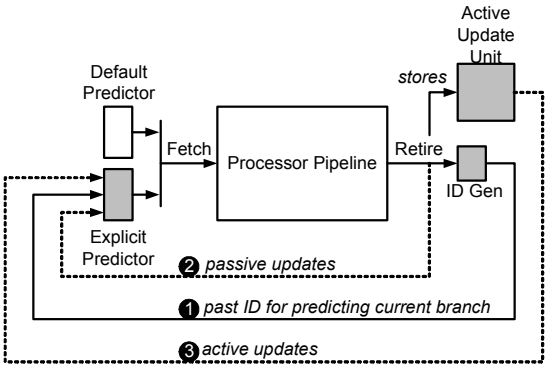


Figure 3. High level view of EXACT.

Figure 3 shows a high level view of EXACT. Instruction fetch is directed by a hybrid predictor, comprised of a default history-based predictor and an explicit predictor (chooser not shown). As branches retire from the processor, their IDs are deduced from producer loads that retired before them (ID Gen). As mentioned above, the explicit predictor predicts the current branch using the ID of a retired branch a fixed distance away (see label 1). The explicit predictor is both passively updated (branches' outcomes are recorded as they retire, see label 2) and actively updated. When a store retires from the processor, its address and value are converted by an active update unit into updates of the explicit predictor (see label 3). Figure 3 shows how the new predictor is not intrusive to the processor pipeline.

For equal cost, a hybrid *gshare+EXACT* predictor yields 60%, 30%, and 27% fewer mispredictions than *gshare* alone, for three misprediction-heavy benchmarks: bzip2, gzip, and twolf, respectively. Similarly, a hybrid *L-TAGE+EXACT* predictor yields

66%, 33%, and 14% fewer mispredictions than *L-TAGE* alone, for bzip2, gzip, and twolf, respectively.

In general, both the explicit predictor and the active update unit need to be large to provide substantial coverage of mispredictions. Fortunately, two crucial properties of EXACT make it practical:

1. *The explicit predictor is scalably pipelinable*: The problem with a large explicit predictor is latency. This component is on the critical fetch path and must provide a prediction each clock cycle. A large branch predictor can be pipelined in a straightforward way if its next index does not depend on immediately preceding predictions [10][19]. The explicit predictor can be pipelined because consecutive indices – which are derived from retired branches' IDs – are independent from the predictor (refer back to Figure 3). While global branch history is optionally included in the index, this is easily accommodated by using an abbreviated index to read out a row of candidate predictions and post-selecting the finalist at the end of the prediction pipeline when the most recent history bits become available [10].
2. *The active update unit is virtualizable*: On the other hand, latency is not an issue for the active update unit, in fact, this feature can actually be exploited to eliminate dedicated storage for this component. A key result is that most benchmarks are tolerant of 400+ cycles of latency to perform active updates, due to the long distances between stores and reencounters with branches that they update. This is precisely the kind of component that predictor virtualization, a technique proposed by Burcea et al. [1], is suited for. The idea is to implement a small level one (L1) version of the component in dedicated storage, backed by a full version in physical memory which can then be transparently cached in higher levels of the general-purpose memory hierarchy (*e.g.*, L2 cache). The advantages include a substantial reduction in dedicated storage, flexible allocation of virtualized active update resources according to application characteristics, and persistence of microarchitectural state.

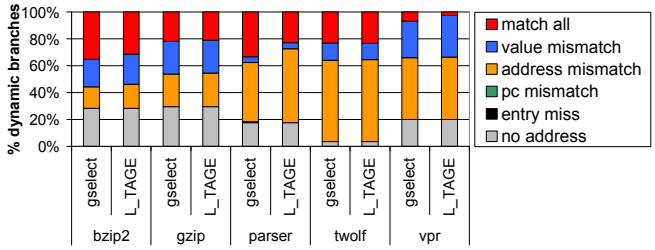
Nevertheless, some designs may favor a small predictor. For example, the leap from an unpipelined or moderately pipelined fetch unit to a deeply pipelined one may be deemed too great, or the area budget for the fetch unit may preclude a large predictor. Accordingly, we also present a more compact form of the explicit predictor. It takes the form of a small cache, which caches only dynamic instances of a static branch that differ from its overall bias. Attaching this small cache to an existing predictor improves accuracy comparably to scaling the existing predictor, but without extending the cycle time appreciably. Adding a 4KB explicit predictor cache and 16KB of other overhead (off the critical-path of instruction fetch) removes 33% of mispredictions from a 4KB *L-TAGE* and 23% of mispredictions from a 8KB *L-TAGE*. These results are comparable to the accuracy of doubling the *L-TAGE* size, but without extending cycle time.

2. CHARACTERIZING MISPREDICTIONS

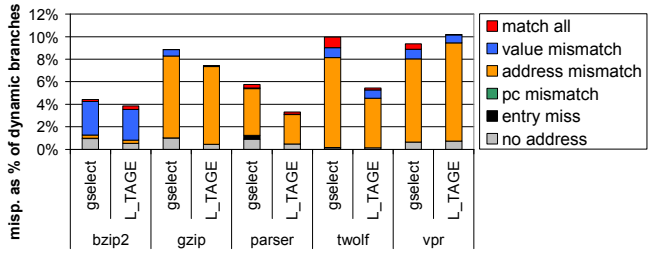
In this section, we characterize mispredictions that escape two global history based branch predictors, *gselect* [14] and *L-TAGE* [18]. We characterized all SPEC2K integer benchmarks but present only five of them, for readability. We present benchmarks that have a misprediction rate of 3% or higher (with a large *L-TAGE* predictor), namely, bzip2, gzip, parser, twolf, and vpr. The *gselect* predictor has a pattern history table (PHT) of 2^{28} entries and the index is formed by concatenating 14 bits of the branch PC with 14 bits of the global branch history register. The *L-TAGE* predictor is composed of 13 predictor components (a simple bimodal component and 12 other partially tagged components) in addition to a loop predictor. Similar to *gselect*, the index for each component is formed by concatenating PC bits with a component-specific amount of folded global history. A geometric series is used to determine global history lengths for each component ranging from 4 bits to 640 bits.

We include additional information in every prediction entry purely for diagnosing the root causes of mispredictions. Every prediction entry is tagged with the ID of the last dynamic branch to have updated the prediction entry. (Other diagnosis information will be introduced later.) As defined in Section 1, the ID of a dynamic branch is the combination of (1) PC of the dynamic branch and (2) addresses of loads on which the dynamic branch depends, which we will refer to simply as "load addresses". The current global branch history is effective context for predicting a given dynamic branch, if the indexed prediction entry is tagged with the dynamic branch's ID.

The graphs in Figure 4 show breakdowns of (a) all branches and (b) just mispredicted branches, as a percentage of all dynamic branches. Each bar is broken down into six components. The "no address" component means the dynamic branch either does not depend on any loads or its outcome is not determined solely by loads. From Figure 4(b), the fact that these dynamic branches contribute a relatively small fraction of mispredictions suggests that global history is effective context for specializing predictions for non-load-dependent dynamic branches. *Gselect* and *L-TAGE* have equal "no address" components because the "no address" component is a property of the program.



(a)



(b)

Figure 4. Breakdown of (a) all branches and (b) mispredicted branches, as a percentage of all dynamic branches.

The "entry miss" component corresponds to accessing a prediction entry for the first time (cold miss). The "pc mismatch" component corresponds to accessing an entry that was last updated by a different static branch (conflict miss/aliasing). Both are negligible.

The "address mismatch" component is of primary interest. In this case, the dynamic branch being predicted differs from the dynamic branch which last updated the prediction entry. They are different

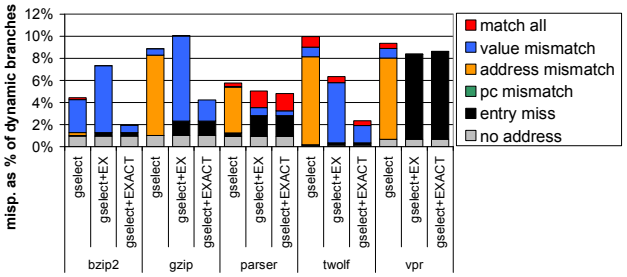
dynamic instances of the same static branch: their PCs match but their load addresses differ. The mismatch means global branch history fails to distinguish the two dynamic branches, therefore, the predictor fails to specialize predictions for them. From Figure 4(a), there is an address mismatch in *gselect* for about 16% (bzip2) to 61% (twolf) of all branch predictions and in *L-TAGE* for about 18% (bzip2) to 61% (twolf) of all branch predictions.

An address mismatch does not necessarily mean the branch will be mispredicted, since different dynamic branches may have the same outcome, only that it is more likely to be mispredicted than if the dynamic branch being predicted were the same as the dynamic branch which last updated the prediction entry. With the exception of bzip2, a large majority of mispredictions is attributed to "address mismatch". Summing up, global branch history does not necessarily distinguish among different dynamic instances of the same static branch, and this is a leading or major contributor to mispredictions in some benchmarks.

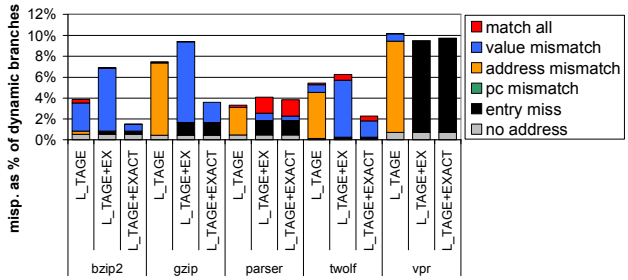
The "value mismatch" component corresponds to the case where the dynamic branch being predicted is the same as the dynamic branch which last updated the prediction entry (their IDs match), but the values at its load addresses were changed by stores since it last updated the prediction entry. To detect this case, each prediction entry is not only tagged with the ID of the dynamic branch that last updated the entry, but also the values contained at its load addresses at that time. A value mismatch does not necessarily mean the branch will be mispredicted, since different values may lead to the same branch outcome, only that it is more likely to be mispredicted than if the values had not changed. From Figure 4(b), "value mismatch" is not a major cause of mispredictions except in the case of bzip2.

On the other hand, mispredictions caused by insufficient specialization ("address mismatch") may hide mispredictions that would otherwise be caused by stores ("value mismatch"). To explore this issue, we use an idealized predictor that explicitly identifies dynamic branches and specializes predictions for them. The predictor is ideal in two ways. First, its size is unbounded. Second, the ID of the dynamic branch is known *a priori* (at the time the branch is predicted). The index is simply the ID of the dynamic branch. The explicit predictor is combined with a global history based predictor (either *gselect* or *L-TAGE*), the latter being used for non-load-dependent dynamic branches.

The graphs in Figure 5 show the breakdown of mispredictions when the explicit predictor is combined with (a) *gselect* and (b) *L-TAGE*. There are three bars for each benchmark. For comparison, the first bar ("gselect" or "L-TAGE") shows mispredictions when only the global history based predictor is used (no explicit predictor). The second bar combines the explicit predictor (EX) with the global history based predictor ("gselect + EX" or "L-TAGE + EX"). The "address mismatch" component of mispredictions is zero because dynamic branches are now provided with dedicated prediction entries. On the other hand, the fraction of mispredictions attributed to "value mismatch" is more substantial now. The results with *gselect* and *L-TAGE* show similar trends. Since *L-TAGE* is more accurate than *gselect* overall, we focus on the results in Figure 5(b). For the "L-TAGE+EX" category, the overall misprediction rate increases compared to just using *L-TAGE*, due to value mismatches supplanting address mismatches as the chief source of mispredictions. In fact, the "value mismatch" component often exceeds the "address mismatch" component that it replaced. There is also an increase in the "entry miss" component for gzip and parser, indicating that a non-trivial fraction of mispredictions are ultimately due to seeing a given dynamic branch ID for the first time; in this study, when there is an "entry miss" in the explicit predictor, the default predictor is used to make the prediction and we do not diagnose the misprediction other than to indicate that it was produced by the default predictor. For vpr, entry misses supplant address mismatches almost one-for-one. In its SimPoint [21], vpr pair-wise compares elements from two large arrays, creating many dynamic branch IDs that are visited only once during the SimPoint. For bzip2, the "value mismatch" component was the chief source of mispredictions originally ("L-TAGE") and its contribution doubles ("L-TAGE + EX"). The third bar in Figure 5 (a) and (b) augments the explicit predictor with active updates by store instructions, using the active-update implementation described later in the paper ("gselect + EXACT" and "L-TAGE + EXACT"). Explicit dynamic-branch prediction and active updates work in concert to substantially reduce the misprediction rate with respect to *gselect* and *L-TAGE*. From Figure 5(b), bzip2, gzip, and twolf, the three benchmarks which experienced the largest increases in "value mismatch" type mispredictions, have many of these mispredictions eliminated by active updates, for a substantial decrease in the overall misprediction rate with respect to *L-TAGE*.



(a)



(b)

Figure 5. Combining the idealized explicit dynamic-branch predictor with (a) *gselect* and (b) *L-TAGE*.

Overall, Figure 5 shows the potential for dramatic reductions in misprediction rates using the two principles of EXACT: explicit dynamic-branch prediction (for achieving desired specialization) and active updates. Notably, except for bzip2, the two techniques are needed in combination: the first technique is needed to eliminate mispredictions caused by insufficient specialization, but in doing so, the predictor is also more vulnerable to stores that require active updates. Bzip2 only requires active updates.

3. EXACT IMPLEMENTATION

Figure 6 shows the major components of the EXACT predictor. A prediction is supplied by either the default predictor (*e.g.*, *L-TAGE*) or the explicit predictor. The explicit predictor is simply a table of 1-bit predictions. The chooser classifies static branches as more suitable for the default predictor or the explicit predictor. The

chooser also singles out static branches that exhibit loop behavior and directs an explicit loop predictor to provide a trip-count to the fetch unit instead of single prediction.

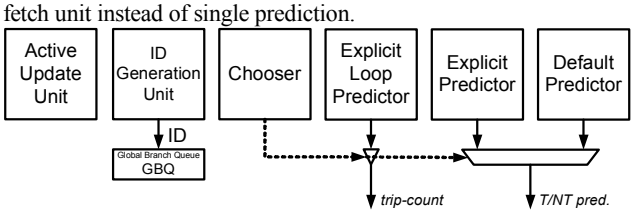


Figure 6. Major components of EXACT.

All components are passively trained as dynamic branches retire from the processor. In addition, both the explicit predictor and the explicit loop predictor may be actively updated by the active update unit. An active update occurs when a store retires from the processor. The ID generation unit observes all instructions as they retire from the processor, in order to propagate load addresses (the basis for IDs) to branches. When a branch retires, its ID is pushed onto the global branch queue (GBQ). The GBQ is used for indexing the explicit predictor and explicit loop predictor.

Sections 3.1 through 3.4 explain ID generation, the GBQ, indexing the explicit predictor, and pipelining the explicit predictor. The explicit loop predictor, chooser, and active update unit are explained in Sections 3.5 through 3.7.

3.1 ID Generation Unit

A non-functional architectural register file (ARF) propagates addresses of loads to branches that depend on them directly or indirectly. In this paper, each logical register in the ARF holds up to four load addresses. Loads write their addresses into their destination registers when they retire from the load queue. ALU instructions propagate addresses from their source registers to their destination registers when they retire from the reorder buffer. Branches obtain their load addresses from their source registers when they retire from the reorder buffer.

To handle registers that are spilled to the stack, we augment the ARF with a small fully-associative cache, called a stack cache. We use a 32-entry stack cache in this paper. Like the ARF, a stack cache entry contains up to four addresses but it is also tagged with a stack address to check for stack-store and stack-load hits. A stack-store copies the addresses contained in its source register from the ARF to the stack cache. If a stack-load hits in the stack cache, the addresses are copied from the stack cache to the stack-load's destination register in the ARF. In summary, load addresses are propagated to branches through both registers and the stack.

A dynamic branch forms its ID by hashing its PC and load addresses together, as follows. The first address is XORed with the second address shifted left by one bit, the third address shifted left by two bits, and so on, for as many load addresses as there are. The result is then ANDed with a mask to extract the low N bits, for an explicit predictor that has 2^N entries. The upper 8 bits of the result is XORed with the lower 8 bits of the PC.

3.2 Global Branch Queue

The GBQ contains IDs of recently retired dynamic branches. When a dynamic branch retires, the ID generation unit pushes its ID onto the GBQ, displacing the oldest ID in the GBQ. GBQ length is discussed in the next section.

3.3 Indexing the Explicit Predictor

We cannot use the ID of a dynamic branch to index the explicit predictor for two reasons. First, typically, its producer loads have not generated their addresses by the time it is fetched. Second, even if addresses were available in time to predict the branch, assembling and associating them with the branch currently being fetched is challenging, whereas the ID generation unit does this straightforwardly when the branch itself retires.

Instead, the index for a branch is based on the ID of a prior retired branch some fixed distance away. The rationale for this approach is that there is repetition in the global sequence of dynamic branches (IDs), due to the program iterating and reiterating over data structures. The chosen distance determines the length of the GBQ. The approach is illustrated in Figure 7 for three scenarios and a distance of 20. The three scenarios differ in how many unretired branches are currently in the processor pipeline, affecting which branch in the GBQ is used to predict the new branch. The third scenario shows what happens when the distance between the new branch and the youngest retired branch in the GBQ is greater than the fixed distance, 20. The problem is that the new branch needs the ID of a branch which has not yet retired. It cannot form an index into the explicit predictor and must use the default predictor instead.

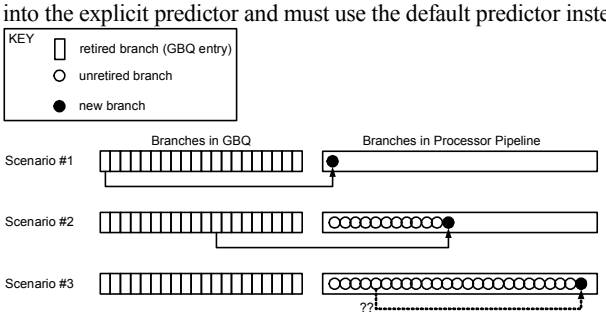


Figure 7. The index for a branch is based on a prior branch's ID some fixed distance away (20 in this example).

This indexing strategy is tantamount to predicting the ID of the current branch from the ID of a distant prior branch. A given ID may lead to any of a number of IDs downstream from it, depending on intervening control-flow (among other things). This is corroborated by our studies which show that hashing the ID of the distant prior branch with global branch history is essential for more closely approximating using the ID of the current branch. We have observed a few exceptions to this general rule. In these exceptional cases, including global branch history may be detrimental because it creates redundant entries in the explicit predictor which has two negative effects: thrashing the predictor and needlessly increasing training time. We concluded that global branch history should be used for some branches and not others. To this end, the chooser – in addition to selecting among the default predictor, explicit predictor, and explicit loop predictor – identifies branches whose indices into the explicit predictor should not include global branch history. When global branch history is included in the index, it is hashed into the low bits of the ID of the distant prior branch.

Figure 8 shows the effect of distance, for two cases: (a) the prior branch whose ID is used for the index is assumed to be retired regardless of distance (*i.e.*, never suffer scenario #3), and (b) whether or not the prior branch is retired is determined through cycle-level processor simulation (may suffer scenario #3).

A distance of 0 means the branch being predicted uses its own ID. For case (a), this yields the lowest misprediction rate but is based on the flawed assumption that its ID is available. Case (b) shows the highest misprediction rate for distance 0 because the default predictor is used almost exclusively. Case (a) shows increasing misprediction rate with distance, with gzip and twolf showing large jumps between distance 0 and 1. This transition is effectively the gap between ideal and real index prediction. Subsequently, misprediction rate increases gradually with increasing distance. Case (b) shows decreasing misprediction rate with distance since increasing distance increases the number of branches predicted by the explicit predictor. Cases (a) and (b) converge in the low to mid 20s for gzip and twolf.

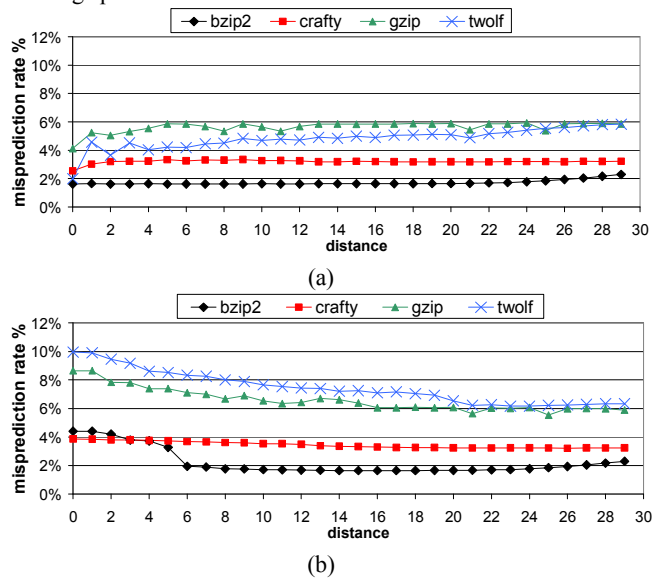


Figure 8. Effect of distance. (a) Prior branch is always retired. (b) Retired status based on processor simulation.

3.4 Pipelining the Explicit Predictor

The explicit predictor is straightforwardly pipelinable because consecutive indices are independent of pending accesses. This is illustrated in Figure 9 for a three-cycle prediction latency.

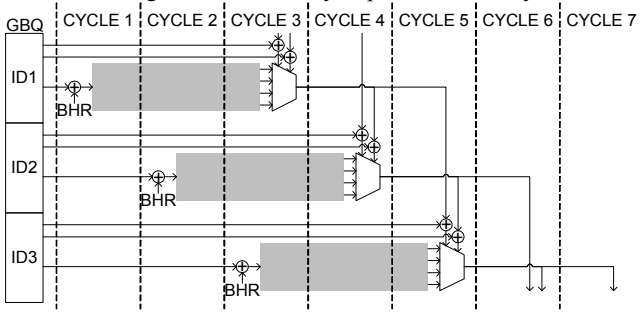


Figure 9. Pipelining the explicit predictor. Diagram shows three consecutive accesses to a three-cycle-latency predictor.

The indices for the first, second, and third accesses are derived from a first ID (ID1), a second ID (ID2), and a third ID (ID3) in the GBQ. Thus, the second access may begin in the second cycle even though the first access is pending, and the third access may begin in the third cycle even though the first and second accesses are pending. Global branch history may be included in the index. The most recent B bits of global history (two bits in our example) are unavailable, however, because they have yet to be produced by preceding pending accesses. This is easily handled by using an abbreviated index that excludes the B unknown bits, reading out a *row* of 2^B candidate predictions, and post-selecting one prediction from among the candidate predictions using the late-arriving B bits at the end of the prediction pipeline [10]. In the example of Figure 9, notice that two bits of both the ID and BHR are omitted from the index, a row of four predictions is read out, and then the missing two index bits are generated at the end of the prediction pipeline (by which time the preceding pending accesses have produced their predictions), which control the MUX.

3.5 Explicit Loop Predictor

Some loops have trip-counts that depend on one or more loads preceding the loop. Applying the same static vs. dynamic framework defined in the introduction, we say that a *static trip-count* translates into multiple *dynamic trip-counts* at run-time. Like a dynamic branch, a dynamic trip-count has an identity (ID) as a whole: the PC of the loop branch combined with the addresses of loads that the dynamic trip-count depends on. Different dynamic trip-counts are distinguished by their IDs. The role of the explicit loop predictor is to provide specialized trip-count predictions to different dynamic trip-counts, analogous to the role of the explicit predictor for dynamic branches.

The explicit loop predictor is accessed using the same index as the explicit predictor (Section 3.3). The explicit loop predictor is managed as a set-associative cache, however, so it can have fewer entries yet still use the full-length indices. The low bits of the index selects a set. Entries within the set are tagged with the remaining high bits of the index. An entry is shown in Figure 10.

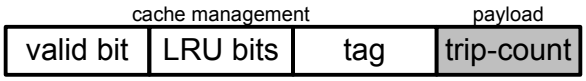


Figure 10. A single entry in the explicit loop predictor.

Dynamic trip-counts are identified as follows. When a backward branch is retired, successive instances of the branch are examined for signature behavior: (1) the branch is taken multiple times in a row followed by a not-taken outcome, thus determining the trip-count value, and (2) all of these dynamic instances of the branch have the same ID. They inherit the same ID because they all test a load-dependent trip-count that was preset outside the loop. It is interesting to note that, without the explicit loop predictor, the explicit predictor alone would be incapable of predicting the dynamic instance that exits the loop since all instances have the same ID (no specialization). Encoding the trip-count (*i.e.*, bundling all iterations into a single prediction) not only provides the immediate benefit of predicting the exit like other loop predictors [4], but also some unprecedented benefits such as (1) explicitly specializing the trip-count for different dynamic data structures and (2) actively-updating the trip-count when these data structures are modified by stores.

3.6 Chooser

The chooser has three jobs: 1) identify branches that are best predicted by the default predictor (§3.6.1), 2) identify branches that exhibit dynamic trip-counts, hence, are best predicted by the explicit loop predictor if it hits or the default predictor otherwise (§3.6.2), and 3) identify branches that do not need global branch history included in their indices (§3.6.3).

The chooser is PC-indexed and is not tagged. Thus, classification is on a per-static-branch basis. This is not only efficient in terms of storage but it also enables rapid classification since all dynamic instances of a static branch train its entry.

3.6.1 Choosing the Default Predictor

Training for this decision occurs in two phases. The first phase is a brief warm-up period during which both the explicit predictor and default predictor are trained for all branches (explicit predictor is not trained for non-load-dependent branches (ID=PC), however). The fetch unit uses predictions solely from the default predictor during the first phase since the chooser is not yet trained. In the second phase, the fetch unit switches to using predictions from the explicit predictor (except for non-load-dependent branches). Meanwhile, training of the chooser proceeds by comparing the ability of the two predictors to predict branches. A chooser entry has a saturating counter and a sticky bit for this purpose. The counter is incremented when the default and explicit predictors are correct and incorrect, respectively; decremented when the default and explicit predictors are incorrect and correct, respectively; and unchanged when both predictors are correct or incorrect. If the counter saturates, the sticky bit is set. A sticky bit of 1 signifies that the default predictor should be used for all dynamic instances of the static branch. Crucially, dynamic instances of this static branch no longer participate in training the explicit predictor or active update unit, dedicating these important resources to more suitable branches.

Certain load-dependent branches are best left for the default predictor for two reasons. First, the indexing strategy may not be accurate for these branches. Recall that this strategy is tantamount to inferring the current branch's ID from a previous branch's ID, a form of address prediction. This may be inaccurate for some branches and they should be weeded out. Second, there may be thrashing in the explicit predictor: explicitly mirroring memory is a worthy cause but requires sufficient predictor capacity, and off-loading some branches to the default predictor may be necessary.

3.6.2 Choosing the Explicit Loop Predictor

Section 3.5 explained how a dynamic trip-count is detected at retirement and how this causes it to be cached in the explicit loop predictor. At the same time, using the PC of the corresponding branch to index the chooser, a sticky bit is set to 1 to indicate that this branch should use the explicit loop predictor if it hits and the default predictor otherwise. If it misses in the explicit loop predictor, the explicit predictor is no better than the default predictor (and maybe worse) since all dynamic instances of the branch have the same ID and predictions cannot be specialized for them individually (discussed at length in §3.5).

3.6.3 Declining Global Branch History

The rationale for including global branch history in the index is that a given ID may lead to any of a number of different IDs downstream from it, depending on intervening control-flow. It is unnecessary, however, if a given ID always leads to the same ID downstream from it. We add a small set-associative cache to detect one scenario or the other (different IDs vs. same ID downstream). It is indexed by ID and the entry contains the previously observed downstream ID. The next time the ID is encountered, if its downstream ID differs from the previously observed downstream ID, then a counter in the chooser (that of the downstream branch's PC) is incremented, otherwise, it is decremented. If the counter saturates at its maximum value, a sticky bit is set indicating to always use global history in the index for the downstream branch. If the counter saturates at its minimum value, a different sticky bit is set indicating to always omit global history from the index of the downstream branch.

3.6.4 Summary of Chooser Contents

Figure 11 summarizes the contents of the two structures in the overall chooser unit. Fields are also annotated with the relevant sections where they were explained.

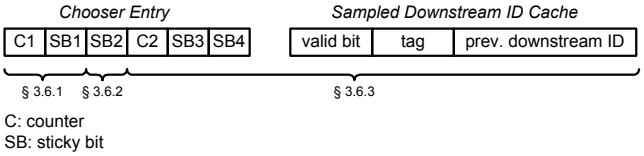


Figure 11. Summary of chooser contents.

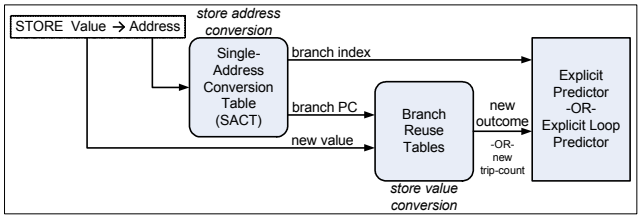
3.7 Active Update Unit

Implementing active updates requires two mechanisms. The first mechanism determines which dynamic branches in the explicit predictor and loop predictor are affected by the store. This process is called *store address conversion* because the store address is converted into affected predictor indices. The second mechanism determines the impact that the store value will have on these dynamic branches in the future. This process is called *store value conversion*, i.e., converting the value into branch outcomes.

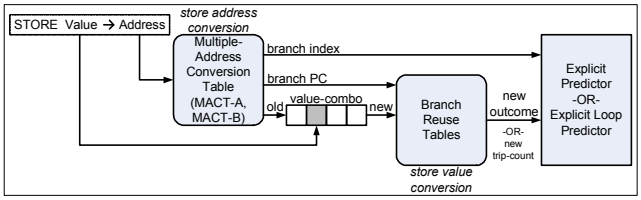
3.7.1 Store Address Conversion

3.7.1.1 Overview

Branches that depend on a single load and branches that depend on multiple loads use different structures for store address conversion, to optimize total cost. Store address conversion – and how it interacts with store value conversion and the predictor – is shown in Figure 12 (a) and (b) for single-address and multiple-address branches, respectively.



(a) for single-address branches



(b) for multiple-address branches

Figure 12. Store address conversion, and its interaction with store value conversion and the predictor.

For single-address branches (Figure 12a), the Single-Address Conversion Table (SACT) outputs two pieces of information for each dynamic branch that depends on the address being stored to: (1) the branch's index in the predictor and (2) the PC of the branch. Store value conversion is based on a novel form of a reuse table [22] which produces a new branch outcome or trip-count using only the branch's PC and the store value. The only value needed is that of the store because the branch depends on only one load. The predictor

can now be actively updated using the branch's index and new outcome or trip-count.

The picture is almost identical for multiple-address branches (Figure 12b), with a key difference. The store value alone is not enough to infer the new branch outcome or trip-count because the branch also depends on the current values at its other addresses (those not being stored to). Thus, the Multiple-Address Conversion Table (MACT), actually composed of two structures MACT-A and MACT-B, outputs two additional pieces of information: (1) the values at all addresses on which the branch depends, called the value-combo, and (2) which value in the value-combo is being updated by the store, called the position. We have determined that the low 16 bits of values suffice. Thus, the value-combo is the concatenation of up to four 16-bit values. The old value-combo is updated by the store and this new value-combo as a whole is the input to store value conversion.

The SACT and MACT are passively trained as dynamic branches retire, by the ID generation unit. Training value-combos requires that the ARF and stack cache not only propagate loads' addresses, but the low 16 bits of their values as well. Also, when a retired store performs an active update, a side-effect is updating the value-combo in the MACT if it exists.

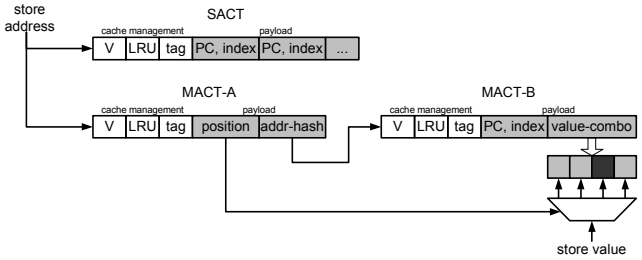


Figure 13. SACT, MACT-A, and MACT-B.

3.7.1.2 SACT

The SACT is a set-associative cache indexed by the store address. The payload of a SACT entry, shown in Figure 13, is a list of {PC, index} tuples, one for each dynamic branch that depends on the address.

3.7.1.3 MACT-A and MACT-B

MACT-A is a set-associative cache indexed by the store address. A dynamic branch occupies multiple entries in MACT-A, one for each of the addresses it depends on, so that stores to any of the addresses can trigger an active update. The payload of a MACT-A entry, shown in Figure 13, consists of (1) the hash of the branch's addresses (addr-hash), *i.e.*, the branch's ID[1] excluding its PC, and (2) the position of the address (which one of up to four addresses) when it was hashed into the ID. The multiple entries occupied by the branch, corresponding to its multiple addresses, will all have the same addr-hash but a unique position. The addr-hash is used to access MACT-B whose payload consists of the other three required items, (1) the branch's PC, (2) the branch's index, and (3) the value-combo, as shown in Figure 13. Thus, all of the branch's MACT-A entries point to the same MACT-B entry for these three items. This reduces cost. Also, by keeping only a single copy of the value-combo in MACT-B instead of replicating it in MACT-A, MACT-A entries perceive each other's updates to the value-combo, which helps if there are multiple stores to different addresses before the next instance of the branch is encountered. This is simply an issue of accuracy and not correctness, however. If one of the branch's MACT-A entries is evicted and there is a store to the evicted entry followed by a store to a resident entry before the branch is reencountered, a partially stale value-combo (first store not included) will be used for the second store's active update. Again, this is only an issue of accuracy. Moreover, flaws are fleeting because MACT-A and MACT-B entries are refreshed by passive updates.

[1] Although more than one dynamic branch may depend on the address, accommodating only one suffices.

3.7.2 Store Value Conversion

The store value (for single-address branches) or store-updated value-combo (for multiple-address branches) is converted to a branch outcome (non-loop branch) or trip-count (loop branch) using novel reuse tables. Both the General Reuse Table (GRT) and Range Reuse Table (RRT) are set-associative caches indexed by the branch's PC.

A GRT entry records the outcome or trip-count that was observed for a given value or value-combo. Each entry can enumerate up to 16 {value/value-combo, outcome/trip-count} pairs, as shown in Figure 14. The reuse test requires a match on the PC and value or value-combo. The GRT is general in that it can be used by any type of branch: non-loop or loop, single-address or dual-address or multiple-address.

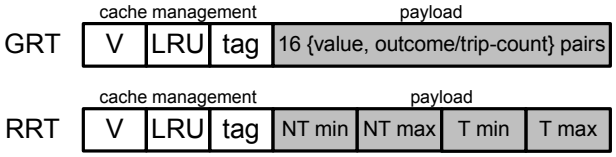


Figure 14. GRT and RRT entries.

The RRT is narrower in scope yet powerful. It can only produce taken/not-taken outcomes (non-loop branches only) and is only intended for single-address and dual-address branches. An RRT entry trains two value ranges: one range that produces a taken outcome and one range that produces a not-taken outcome. Using ranges has two advantages over enumerating distinct values. First, only four values need to be recorded for each static branch (maximum and minimum for taken outcomes and maximum and minimum for not-taken outcomes) instead of recording every distinct value seen by the branch, as shown in Figure 14. Second, an active update can be performed even for a value that has not been seen by the branch, if it falls within either the taken or not-taken ranges.

For single-address branches, the RRT reuse test checks for a match on the PC and for the store value to fall within one of the ranges. For dual-address branches, the two 16-bit values in its store-updated value-combo are subtracted and the reuse test checks for this result to fall within one of the ranges. The intuition behind this heuristic is that, often, this type of branch compares two load-dependent values.

If, while training an RRT entry, one range is updated such that it overlaps the other range, the static branch is evicted from the RRT and added to the GRT. Its existence in the GRT prevents its further use of the RRT. This action adapts to the reality that using ranges for the branch is probably unreliable.

4. METHODOLOGY

All results in this paper are based on custom predictor and processor simulators derived from the SimpleScalar toolset [2]. Eleven of the integer SPEC2K benchmarks were used with reference inputs. We compiled these benchmarks to the SimpleScalar PISA instruction set using the SimpleScalar gcc-based compiler with –O3 optimization. The eon benchmark did not compile. The SimPoint toolset [21] was used to locate representative simulation points. The simulator skips to the SimPoint minus 10 million instructions, warms up for 10 million instructions, and then simulates for 100 million instructions. Parameters of the modeled processor are shown Table 1.

Table 1. Microarchitecture parameters.

L1 I&D Caches	64KB, 4-way, 64B line, hit=1 cycle, miss=10 cycles, 32 MHSRs
L2 Cache	Unified, 2MB, 8-way, 128B lines, hit=10 cycles, miss=200 cycles, 64 MHSRs
Reorder Buffer	256
Issue Queue	64
Load-Store Queue	64
Rename Map Checkpoints	16
Fetch-to-exec. pipe depth	20 stages
Fetch/Issue/Retire Width	4 instr./cycle

All results are presented in the context of cycle-level processor simulation to model the effect of branch distance in forming the predictor's index. Throughout, the GBQ length is 21, therefore, the index is formed using the ID of the 21st branch away. If this branch is not yet retired, the default predictor is used. The index includes either 0 or 6 bits of global branch history: the chooser guides this decision on a per-static branch basis. A warm-up period of 10 million instructions is used, after which the chooser commences training and hybrid prediction is engaged.

We divide EXACT's subcomponents into two classes. Subcomponents in the first class can be scaled up in size with continuing effect on reducing misprediction rate, therefore, these are studied in Section 5. They include: (1) the default predictor, (2) the explicit predictor, and (3) the SACT component of the active update unit. Subcomponents in the second class are no less critical but their sizes are not as closely coupled to a program's large data structures, hence, scaling them up has negligible effect. The configurations of subcomponents in the second class are fixed and shown in Table 2. The configuration of each subcomponent was arrived at by unconstraining all other subcomponents and finding the point of diminishing returns. The total cost of these fixed subcomponents is 18.6 – 19.7 KB, depending on N (the number of bits in the ID and index).

5. RESULTS

5.1 Impact of Real Indexing

In this section, we measure the impact of real indexing in the context of unconstrained resources: the unconstrained *gselect* and *L-TAGE* predictors of Section 2 and unconstrained EXACT subcomponents. Figure 15 shows misprediction rates of each default predictor alone (*gselect* or *L-TAGE*), each default predictor with ideal-indexed EXACT, and each default predictor with real-indexed EXACT. Ideal indexing is where the ID of the branch is used. Real indexing is where the ID of the 21st prior branch is used (plus history if chosen) if it has been retired.

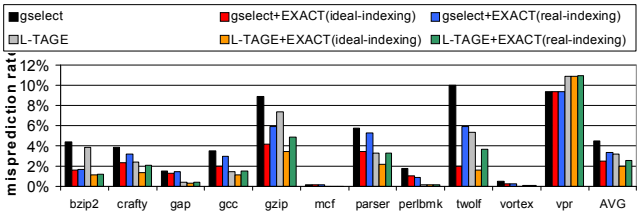


Figure 15. Misprediction rates with large predictors.

From Figure 15 and earlier characterizations from Section 2, we classify the eleven benchmarks into three categories. The first category consists of benchmarks that do not benefit from EXACT, either because they already have impressive accuracy (gap, mcf, perlbmk, vortex), particularly with *L-TAGE*, or there is no repetition of dynamic branches / IDs (vpr). For these benchmarks, EXACT does not improve or appreciably degrade the accuracy for both ideal and real indexing. The second category consists of benchmarks that have moderate misprediction rates, show some improvement with ideal indexing, but very slightly improve or degrade with real indexing (crafty, gcc, parser). The third category consists of benchmarks that show an impressive reduction in misprediction rate with ideal indexing and attain a considerable portion of this reduction with real indexing (bzip, gzip, twolf). On average, EXACT with ideal indexing removes 44% and 37% of mispredictions in *gselect* and *L-TAGE*, respectively, while EXACT with real indexing removes 26% and 20% in *gselect* and *L-TAGE*, respectively.

Table 2. Fixed-configuration subcomponents. Not included: default predictor, explicit predictor, and SACT.

Unit	Structure	# entries / organization	Contents per entry	Size (KB)
ID Generation Unit	ARF	67 entries	4 valid bits + 4 20-bit addresses + 4 16-bit values	1.21
	stack cache	32 entries/fully-assoc	1 valid bit + 5-bit LRU + 20-bit tag + *same payload as ARF entry*	0.68
GBQ		21 entries	N-bit ID	0.04 – 0.06
Explicit Loop Pred.		256 entries/8-way assoc	1 valid bit + 3-bit LRU + (N-5)-bit tag + 8-bit trip-count	0.75 – 0.94
Chooser	Chooser Table	1024 entries/not tagged	5-bit C1 + 1-bit SB1 + 1-bit SB2 + 9-bit C2 + 1-bit SB3 + 1-bit SB4	2.25
	downstr. ID $	512 entries/8-way assoc	1 valid bit + (N-6)-bit tag + N-bit prev. downstream ID	1.69 – 2.44
Active Update Unit	RRT	128 entries/4-way assoc	1 valid bit + 2-bit LRU + 9-bit tag + 4x16-bit for ranges	1.19
	GRT	32 entries/4-way assoc	1 val. bit + 2-bit LRU + 11-bit tag + 16x64-bit values + 16x8-bit trip-counts	4.55
	MACT-A	512 entries/16-way assoc	1 valid bit + 4-bit LRU + 15-bit tag + 2-bit position + 20-bit addr-hash	2.63
	MACT-B	256 entries/16-way assoc	1 val. bit + 4-bit LRU + 16-bit tag + 14-bit PC + N-bit index + 64-bit combo	3.59 – 3.78
Total cost of fixed subcomponents (does not include default predictor, explicit predictor, and SACT):				18.6 – 19.7

Notes:
1. Regarding number of entries in ARF: PISA ISA has 32 integer registers, 32 floating-point registers, and 3 other registers (hi, lo, fcc).
2. N = # bits in the ID = # bits in the index. N is varied between 16 and 22. This introduces variation in the total cost of the fixed-configuration subcomponents, as shown.
3. All addresses are truncated to 20 bits after discarding the low 1, 2, or 3 bits for halfwords, words, and doublewords, respectively. Likewise, the addr-hash is 20 bits. When N>20, the upper few bits of the ID consists of only PC bits.
4. All PCs are truncated to 14 bits after discarding the low 3 bits (PISA instructions are 8 bytes).

5.2 Impact of Active Update Latency

In this section, we measure the impact of active update latency in the context of the unconstrained *gselect+EXACT(real-indexing)* from the previous section. Figure 16 shows misprediction rate as a function of active update latency in increments of 50 cycles. The latency is from the cycle that the store retires to the cycle when the updated outcome or trip-count is reflected in the explicit predictor or loop predictor. The three benchmarks that benefit substantially from EXACT are included as well as crafty which mildly benefits. Gzip is insensitive to latency. Twolf is very tolerant of latency up to 400 or 500 cycles. The change in its misprediction rate from 0 to 400 cycles is 6.26% to 6.39%. At around 500 cycles, it becomes more sensitive.

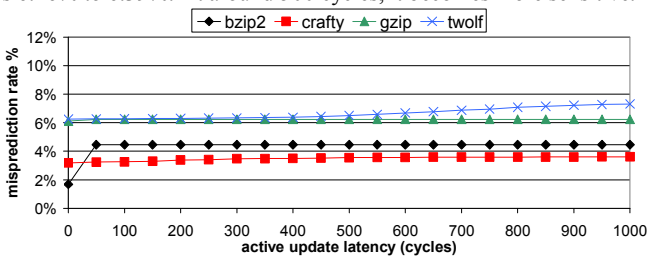


Figure 16. Misprediction rate vs. active update latency.

5.3 Accuracy vs. Storage Budget

This section provides results for large predictors used in leading-edge processing cores, where the issue of predictor access latency is relieved by means of pipelining. EXACT is a good solution when scaling the conventional branch predictor does not provide any accuracy improvement.

The graphs in Figure 17 show misprediction rate as a function of cost using the methodology described in Section 4: the subcomponents in Table 2 have fixed configurations with a total cost around 19KB and only the default predictor, explicit predictor, and SACT configurations are varied. For each cost point, resources are allocated to these three subcomponents based on design space exploration that yields the lowest misprediction rate. With cost constrained, we substitute *gselect* with *gshare* [14]. The *gshare* index for each cost point (*e.g.*, length of global branch history register) is also based on exploration.

We explore two implementations of the SACT:

1. The first uses dedicated storage. In this case, the SACT is included in the design space exploration (*i.e.*, its size is varied) and its entire cost is included in the cost comparison. In Figure 17, grey points correspond to the SACT being implemented in dedicated storage (labeled *gshare+EXACT* or *L-TAGE+EXACT*).
2. The second uses a small, dedicated L1 SACT combined with a larger, virtualized L2 SACT, applying the concept proposed by Burcea et al. [1]. Both are fixed in size. The L1 SACT consumes 10KB of dedicated storage, which is added to the original fixed cost of 19KB for an adjusted fixed cost of 29KB. The L2 SACT is pinned in physical memory [1]. A whole SACT set is 80 bytes, which fits entirely within a single L2 cache block. Thus we align SACT sets at block boundaries. We fix the size of the L2 SACT to be 512 KB. This cost is not included in the cost comparison, but the virtualized L2 SACT contends with the application's instructions and data for L2 cache resources, which is relevant for performance results presented in §5.4. In Figure 17, black points correspond to the virtualized SACT (labeled as *gshare+EXACT (VIRT. SACT)* or *L-TAGE+EXACT (VIRT. SACT)*).

5.3.1 Dedicated Storage for SACT

The first cost point where *L-TAGE+EXACT* overtakes *L-TAGE* is 40KB for bzip2 (1.8% down from 3.8%), 353KB for gzip (6.7% down from 7.2%), and 155KB for twolf (7.1% down from 7.3%). Gzip's sharp corner is due to its working set fitting in the SACT, which is the component that expands the most as more resources become available. Also notable about gzip is that *gshare+EXACT* overtakes *L-TAGE* at the 525KB cost point (6.3% down from 7.2%). At a close cost point of 533KB, *L-TAGE+EXACT* reaches 5.4%, nearly 2 points lower than *L-TAGE*. Bzip2 and gzip are perfect examples in which scaling the conventional branch predictor (*gshare* or *L-TAGE*) does not yield any accuracy improvement, whereas EXACT can easily yield benefits from scaling resources. For twolf, *L-TAGE* is able to capitalize on more resources for a fairly broad cost range. So does *L-TAGE+EXACT*, improving *L-TAGE*'s misprediction rate by 0.2 points at 155KB to 0.5 points at 667KB, with the gap widening further at subsequent cost points.

5.3.2 Virtualized SACT

In Figure 17, the curves qualified with the label "(VIRT. SACT)" correspond to a virtualized L2 SACT. For bzip2, the virtualized implementation is slightly costlier than the dedicated one, since the L1 SACT is actually larger than needed (curve is slightly shifted to the right). Gzip benefits significantly from a large, virtualized L2 SACT: it sheds 200-300KB of dedicated storage for the same accuracy. For twolf, the virtualized L2 SACT only moderately improves its accuracy at a given cost point.

5.4 Explicit Predictor Cache (EX-cache)

Some designs may favor a small predictor, for example, the leap from an unpipelined or moderately pipelined fetch unit to a deeply pipelined one may be deemed too great, or the area budget for the fetch unit may preclude a large predictor. We present a more compact form of the explicit predictor to target the domain of smaller predictors.

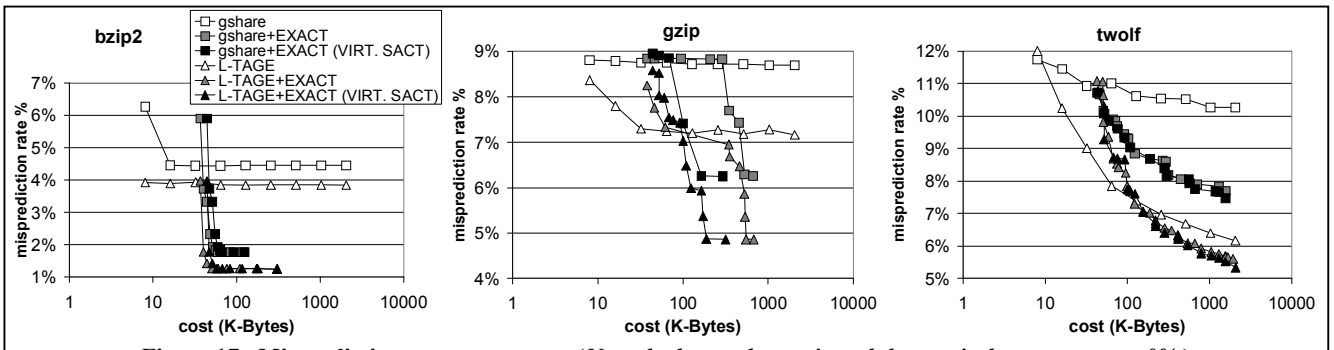
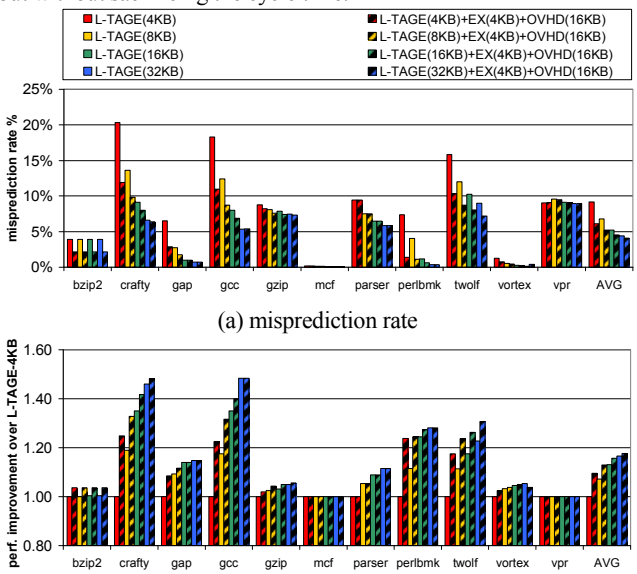


Figure 17. Misprediction rate versus cost. (*Note the log-scale x-axis and the y-axis does not start at 0%.*)

First, the following features are simplified or removed: (1) the ARF only propagates a single address and value; (2) the stack cache is removed (no dependency-propagation through the stack); (3) the explicit loop predictor is removed; (4) all branches include history in their indices, so the downstream ID cache is removed and the chooser is simplified by removing extra bits relating to it; (5) the MACT-A and MACT-B tables are removed (since we only propagate one address in the ARF, the SACT alone will actively update all branches); (6) the GRT will no longer need trip-counts since the explicit loop predictor is removed. This cuts down the total cost of the fixed subcomponents (previously shown in Table 2) from about 20KB to 4KB. In the results that follow, we use an L1 SACT of 12KB with a virtualized L2 SACT in memory.

Second, we modified the explicit predictor component to control which predictions are provided by it. We use a tagged table to cache predictions, called the EX-cache. A dynamic branch's index into the predictor is based on the same information as before (past retired branch's ID and global history), except now only the lower part of this index is used to access a set in the EX-cache and the upper part of the index forms the tag. To reduce the pressure on the EX-cache coming from the huge number of dynamic branches, the EX-cache caches only dynamic branches that have the uncommon branch outcome for a certain branch PC. To do so, we add a 3-bit saturation counter in the chooser table for each branch PC (similar to a bimodal predictor). The branch's bias (indicated by the saturation counter) is checked when the EX-cache is passively trained at retirement. If the current branch outcome is the same as the branch's bias in the chooser, the dynamic branch is not inserted in the EX-cache if it is not cached or evicted if it is cached. But if the current branch outcome differs from the branch's bias, then the dynamic branch is inserted in the EX-cache (if it is not already cached). When making a prediction at fetch-time, the EX-cache is accessed in parallel with the chooser: if there is a hit in the EX-cache, then the prediction is opposite that of the branch's bias in the chooser; if there is a miss in the EX-cache, then the prediction is that of the branch's bias. Active updates are handled in a similar fashion: if the active-update outcome matches the branch's bias, then the index being updated is removed from the EX-cache; if the active-update outcome differs from the branch's bias, then the index being updated is inserted in the EX-cache. In the latter case, since the SACT caches {PC, index} tuples influenced by a certain address, it has the index that must be inserted into the EX-cache. This methodology requires the SACT to be trained with all branches that use the EX-cache regardless of their outcome. Again, this is efficiently accommodated by means of the virtualized L2 SACT. The EX-cache requires only tag arrays without any data arrays to cache predictions since the prediction is decided using hit/miss in the EX-cache and the bias bit in the chooser.

The simplified EXACT is comprised of a 4KB EX-cache and 16KB of other fixed-cost overhead (including the L1 SACT). Note that only the 4KB EX-cache and 1KB chooser table reside on the critical path of instruction fetch, the other overhead is off the critical path which does not affect the cycle time of the processor. Figure 18 shows both accuracy and performance improvement achieved by adding a 4KB EX-cache and 16KB overhead to different *L-TAGE* size configurations. For the two benchmarks bzip2 and twolf, adding the 4KB EX-cache and 16KB overhead improve on accuracy and performance compared to doubling the *L-TAGE* predictor size. Gzip benefits less from the simplified EXACT compared to results from the previous section. This is due to the large working set of gzip that needs a larger EX-cache to capture all dynamic branches (it benefits less from the bias optimization than other benchmarks). On average, adding the simplified EXACT provides a significant improvement in accuracy and performance that is comparable to doubling *L-TAGE* but without sacrificing the cycle time.



(a) misprediction rate

(b) performance improvement

Figure 18. Impact of adding a 4KB EX-cache and 16KB overhead to different L-TAGE predictor sizes.

6. RELATED WORK

With the advent of two-level adaptive branch prediction [25], there has been a plethora of research on branch predictors that combine branch PCs, local/global branch history, and path information in ingenious ways to achieve ever higher accuracy. For brevity, we focus instead on closely related work that recognizes the need to sometimes correlate on program values explicitly [3][6][7][8].

Gonzalez and Gonzalez [7] explicitly value-predict the source operands of a branch to calculate its direction early in the pipeline. Heil et al. [8] proposed using the last committed difference between a branch's source operands coupled with the number of outstanding instances of the branch.

Chen et al. [3] proposed using live-in register values of a dynamic branch's backward-slice to predict the branch, if these values are available in the register file (committed). Backward-slices terminate at loads. Their results showed that 80% of dynamic branches depend on pending loads whose values are unavailable in the pipeline for making predictions. This highlights the need for explicitly predicting load values or addresses. Our real indexing strategy predicts the ID of the dynamic branch being fetched which is tantamount to predicting the addresses of loads in its backward-slice.

Gao et al. [6] developed the ABC (address-branch correlation) predictor specifically for hard-to-predict branches that depend on loads that miss in the L2 cache. They exploit two observations, (1) the value contents of the data structures tested by these branches tend to be stable, therefore, a branch outcome correlates well with simply the address of the data structure, and (2) while the actual value is unavailable by virtue of being retrieved from the memory system, the address is available since the load on which the branch depends has already issued to the memory system. Accordingly, they

use the address of the missed load to repredict the direction of the load's dependent branch. The fetch unit is redirected if the reprediction does not match the original prediction. Repredicting long-latency branches is comparatively simpler than the topic tackled in this paper because the address is available for the branch being repredicted, or, for linked-data-structure type loads, the address of a previous iteration of the same load is available. In contrast, our predictor must hide the core pipeline latency for all branches, requiring the ID (essentially load addresses) for every branch to be predicted which is the impetus for our novel indexing strategy. Moreover, our paper is the first to propose active updates and show that active updates are important when targeting all branches: this result differs from Gao et al.'s observation that underlying data is stable, which is a reasonable assumption when narrowly targeting some mispredicted branches.

Thomas et al. [24] and Sazeides et al. [17] used the program's dataflow graph to identify branches that are correlated, with the goal of pin-pointing non-consecutive global history bits that are correlated. Paring down global history to its useful bits is good for reducing training time and making a small predictor perform close to a large one. Their work is still confined to the accuracy bounds of branch-history-based prediction, which we show is limited by lack of specialization and store updates.

7. SUMMARY

State-of-the-art branch predictors have impressively pushed the envelope of what is possible with global branch history. This paper identified two interrelated problems that challenge microarchitects to move forward with fundamental changes in branch predictor design. The first problem is not sufficiently specializing predictions for memory dependent branches and the second problem is the inability to aggressively adapt to stores even if predictions are explicitly specialized for these dynamic branches. The two principles of EXACT are along these lines: explicitly specialize predictions for these dynamic branches and actively update them as stores occur.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported by NSF grants CCF-0702632 and CCF-0916481, SRC grant 2007-HJ-1594, and an Intel gift. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] I. Burcea, S. Somogyi, A. Moshovos, B. Falsafi. Predictor Virtualization. *ASPLOS-XIII*, 2008.

[2] D. Burger, et al. Evaluating Future Microprocessors: The Simplescalar Toolset. CS-TR-96-1308, Univ. of Wisc., 1996.

[3] L. Chen, et al. Dynamic Data Dependence Tracking and its Application to Branch Prediction. *HPCA-9*, 2003.

[4] M. de Alba and D. Kaeli. Path-based Hardware Loop Prediction. *CICINDI-4,* 2002.

[5] F. Gabbay and A. Mendelson. The Effect of Instruction Fetch Bandwidth on Value Prediction. *ISCA-25*, 1998.

[6] H. Gao, et al. Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches.*HPCA-14*, 2008.

[7] J. Gonzalez, et al. Control-Flow Speculation through Value Prediction for Superscalar Processors. *PACT*, 1999.

[8] T. Heil, Z. Smith, J. E. Smith. Improving Branch Predictors by Correlating on Data Values. *MICRO-32*, 1999.

[9] M. D. Hill, M. R. Marty. Amdahl's Law in the Multicore Era, *IEEE Computer*, 2008.

[10] D. Jiménez. Reconsidering Complex Branch Predictors. *HPCA-9*, 2003.

[11] D. Jiménez, C. Lin. Dynamic Branch Prediction with Perceptrons. *HPCA-7*, 2001.

[12] R. Kumar, et al. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. *PACT-15*, 2006.

[13] G. H. Loh, D. S. Henry. Predicting Conditional Branches With Fusion-Based Hybrid Predictors. *PACT-11*, 2002.

[14] S. McFarling. Combining Branch Predictors. DEC WRL TN-36, 1993.

[15] P. Michaud, et al. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *ISCA-24*, 1997.

[16] T. Y. Morad, et al. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE CAL*, 5(1), 2006.

[17] Y. Sazeides, et al. The Significance of Affectors and Affectees Correlations for Branch Prediction. *HiPEAC-3*, 2008.

[18] A. Seznec. The L-TAGE Branch Predictor. *JILP*, 2007.

[19] A. Seznec and A. Fraboulet. Effective Ahead Pipelining of Instruction Block Address Generation, *ISCA-30*, 2003.

[20] A. Seznec, P. Michaud. A case for (partially) TAgged GEometric history length branch prediction. *JILP*, 2006.

[21] T. Sherwood, et al. Automatically Characterizing Large Scale Program Behavior. *ASPLOS-X*, 2002.

[22] A. Sodani and G. Sohi. Dynamic Instruction Reuse. *ISCA-24*, 1997.

[23] M. A. Suleman, et al. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. *ASPLOS-14*, 2009.

[24] R. Thomas, et al. Improving Branch Prediction by Dynamic Dataflow-Based Identification of Correlated Branches from a Large Global History. *ISCA-30*, 2003.

[25] T.-Y. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. *ISCA-19*, 1992.

[26] The 2nd JILP Championship Branch Prediction Competition (CBP-2). http://camino.rutgers.edu/cbp2/