A Case for a Software-Managed Reconfigurable Branch Predictor

Muawya Al-Otoom
Intel Corporation
muawya.m.al-otoom@intel.com

Rami Sheikh, Eric Rotenberg
North Carolina State University
rmalshei@ncsu.edu, ericro@ncsu.edu

ABSTRACT

Workloads present diverse performance challenges. Consequently, many proposed microarchitecture techniques apply to only a subset of applications. This limits their appeal, as the added resources tend to be unused by the remaining applications. Designing reconfigurable hardware is one way to tackle this issue. In this paper, we propose EXACT-S, a software-managed reconfigurable branch predictor that targets workloads with hard-to-predict load-dependent branches encountered in the context of sequencing arrays and linked-lists. EXACT-S is inspired by the previous EXACT predictor and targets its limitations. EXACT is indexed by branches' load addresses and actively updated by stores. With EXACT-S, the idea is to have a light-weight run-time layer to convey key information directly to the fetch unit that it can use to generate branches' load addresses in a timely manner. This approach is more accurate because it uses branches' load addresses directly rather than prior branches' load addresses. Moreover, with regard to active updates, there is no need for a large table to convert store addresses to predictor indices because of the direct indexing strategy. As a result, EXACT-S is simpler, less expensive and more accurate than EXACT. For applications that suffer poor prediction accuracies due to load-dependent branches, EXACT-S removes up to 50% of their mispredictions. For all other applications, the proposed reconfigurable predictor relinquishes EXACT-S storage to the base L-TAGE predictor, thus achieving the same prediction accuracy as a similarly-sized fixed L-TAGE predictor.

1. INTRODUCTION

High performance processors have matured to the point that it is rare nowadays to conceive of a single microarchitecture technique that will speed up all applications. All of the "low hanging fruits" – techniques that target fairly universal behavior – have been implemented, and are being scaled out (e.g., OOO execution). Now, different applications have specific behaviors that can only be effectively addressed by targeted solutions.

This situation has hindered microarchitecture research. Solutions tend to benefit a subset of applications, usually yielding the proposed hardware unused by the remaining applications. This makes it difficult to justify their inclusion in a commercial processor, stifling innovation.

One way to address this issue is by using reconfigurable hardware that releases the unused resource to be allocated in a way that benefits the remaining workloads. As an example, we propose a reconfigurable branch predictor that targets branches that traverse large data structures. This class of branch has been shown to trouble even sophisticated branch predictors [1] but is not present in all workloads.

In a conventional branch predictor, the predictor index is based on local or global branch history patterns. The patterns used to predict different instances of the branch are often the same [1]. In the context of data structure traversal, this means that different instances of the branch, corresponding to different elements of the data structure are likely to end up sharing entries in the predictor table. Unfortunately, if instances sharing an entry have different outcomes, they will be mispredicted. Note that this sharing behavior is a result of the context used to predict branches, not limited storage.

To fix this problem, the recently proposed auxiliary predictor, EXACT [1], combines the branch PC with the address of the element being tested to provide a unique branch ID. More formally, prior work defined the ID of a branch instance to be the program counter (PC) of the branch hashed together with the address of the load that the branch depends on [1]:

ID = hash(branch's PC, branch's load address)

Ideally, the index of the branch instance is its ID (truncated to the number of index bits). Unfortunately, the ideal indexing strategy described above is difficult to achieve in practice, because the load that a branch depends on is unlikely to have generated its address before the branch is fetched. To deal with this problem, the authors of EXACT proposed indexing the predictor with a prior, retired branch's ID some fixed distance away (they used the 21st branch away in their final experiments) instead of the current branch's ID, which is unavailable. Using a proxy branch to predict the current branch works reasonably well due to repetition in the global sequence of branches and their IDs. Unfortunately, there are key drawbacks of EXACT's indirect indexing strategy:

1- *Squandering accuracy potential*: The accuracy of indexing with the current branch's ID is impressive, but at best indexing with a prior branch's ID achieves half of this potential and sometimes less [1]. Using a prior branch's ID as a proxy for the current branch's ID is a subtle form of load address prediction (predicting the current branch's load address from a prior branch's load address), and is imperfect.

2- *Redundant predictor entries*: It was found that some recent global branch history bits need to be included in the index of certain branches [1]. This is because a prior branch may lead to multiple alternative downstream branches along different control-flow paths. Using only the prior branch's ID causes the alternative branches to have the same index and share a predictor entry. Including history ensures the alternative branches have dedicated predictor entries but also creates redundant entries if some history bits do not influence the alternatives. Redundant entries increase pressure on the available storage. As a result, a larger predictor is needed than if the current branch's ID were used to index the predictor.

3- *Cost and complexity of "Active Updates"*: A store to an element of a data structure may cause its corresponding branch outcome to flip when it is next encountered. With conventional passive updates, a misprediction is suffered before the predictor is retrained. On the other hand, EXACT is unique in that stores update the predictor (referred to as "active updates") [1]. An unfortunate but necessary side-effect of the indirect indexing strategy is that a branch's index into the predictor cannot be inferred from the load address on which it depends. This leads to a complex and storage-intensive active update unit: a large table is needed to convert store addresses to the predictor indices that must be updated. The dedicated storage can be reduced by virtualizing the large table in memory, but the virtualized table occupies cache space and is complex to manage.

In this paper, we propose the EXACT-S predictor. EXACT-S has two unique aspects. First, it is *reconfigurable*. Workloads that suffer poor branch prediction accuracies due to traversing data structures will benefit from EXACT-S's unique ID-based indexing. For all other workloads, EXACT-S resources are diverted to the base predictor, so that its resources benefit all workloads.

Second, it is *software-managed*. A light-weight run-time layer conveys key information directly to the fetch unit that it can use to generate branches' load addresses in a timely manner. Thus, the auxiliary predictor can be indexed directly with the current branch's ID. This allows for eliminating the limitations of EXACT [1] discussed above. First, EXACT-S is more accurate because it uses branches' IDs directly rather than prior branches' IDs. Second, the direct indexing strategy translates to zero redundancy in the predictor. Third, active updates are simple and inexpensive because there is no need for a large table to convert store addresses to predictor indices since the affected branch's index can be inferred from the store address.

Since EXACT-S is under software control, there is no need for a hardware chooser to learn which branches should be predicted using the ID-based indexing vs. the history-based indexing. This eliminates both hardware (reducing complexity, cost and power) and training time (performance overhead) with respect to EXACT.

Applying EXACT-S to SPEC2K integer benchmarks revealed two perfect candidates that benefited from the unique ID-based indexing, namely gzip and twolf, where 33% and 50% of mispredictions are removed, respectively (compared to a similarly-sized L-TAGE predictor). For other applications, the unused predictor storage is absorbed by the default predictor via our reconfigurable design: the reconfigured predictor shows similar accuracy to a similarly-sized fixed L-TAGE predictor.

2. EXACT versus EXACT-S

In this section, we first review the operation of the original EXACT predictor (Section 2.1) and then provide an overview of the proposed EXACT-S, its operation, and its simplifications with respect to EXACT (Section 2.2).

2.1 Background on EXACT

Figure 1 shows a high level view of EXACT [1]. Instruction fetch is directed by a hybrid predictor, comprised of a default history-based predictor, an auxiliary predictor indexed by ID called the *explicit predictor*, and a chooser. As branches retire from the processor, their IDs are deduced from producer loads that retired before them (ID Gen) and the IDs are pushed into the Global Branch Queue (GBQ). As alluded to previously, the explicit predictor predicts the current branch using the ID of a retired branch a fixed distance away. This indirect indexing strategy is facilitated by the GBQ (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).

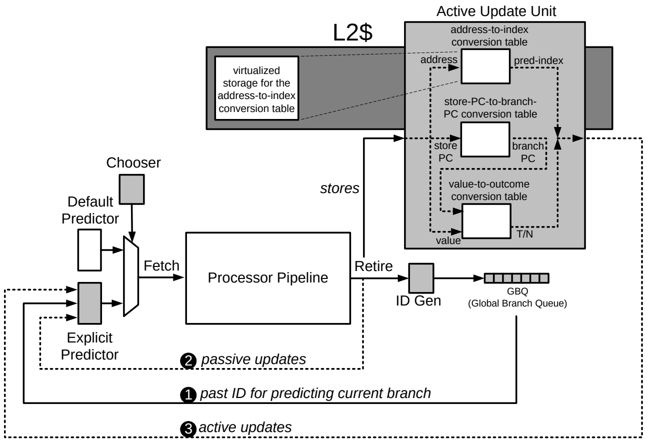

Active Update Unit
L2$
address-to-index conversion table
virtualized storage for the address-to-index conversion table
address
pred-index
store-PC-to-branch-PC conversion table
store PC
branch PC
value-to-outcome conversion table
value
T/N
stores
Chooser
Default Predictor
Fetch
Processor Pipeline
Retire
ID Gen
GBQ (Global Branch Queue)
Explicit Predictor
2 passive updates
1 past ID for predicting current branch
3 active updates

Figure 1. High-level view of EXACT.

Two mechanisms are required for active updates: 1) converting the store address into a predictor index to be updated, and 2) converting the store value into a branch outcome.

Address-to-index conversion would be straightforward if the index was based on the branch's ID: the index would be the store address hashed with the branch's PC. To obtain the branch's PC, EXACT learns which static stores affect which static branches by training a small store-PC-to-branch-PC conversion table (Figure 1).

Unfortunately, the branch's index is based on some other branch's ID. This means its index cannot be determined from the store address directly. Instead, a large address-to-index conversion table is required. The table records, for each address, the explicit predictor's indices corresponding to branches that depend on that address.

Converting the store value to a branch outcome is achieved by the value-to-outcome conversion table (Figure 1), which is a compact reuse table that remembers for each static branch two ranges of load values, one range that causes it to be taken and the other that causes it to be not-taken. We call this table the Range Reuse Table (RRT). The inputs to this table are the branch's PC (produced by the store-PC-to-branch-PC conversion) and the store value.

The store-PC-to-branch-PC and value-to-outcome conversion tables are small in size since they record static branch information unlike the address-to-index conversion table that records information about dynamic branches.

The amount of dedicated storage required for the address-to-index conversion table is reduced by virtualizing it [3]. 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). A potential drawback of virtualization is that it significantly increases the worst-case latency for performing a single active update. Latency is not an issue for the active update unit, however, because 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. Virtualization of the address-to-index conversion table is depicted in Figure 1.

2.2 Overview of EXACT-S

Figure 2 shows a high-level view of EXACT-S. EXACT-S has the same high-level structure as EXACT but is simpler.

EXACT-S exploits software intervention to make the indexing strategy both more accurate and more efficient. In most of the applications that benefit from EXACT, many of the data structures tested by branches are arrays and to a lesser extent linked-lists (which EXACT-S handles as logical arrays as will be discussed in a later section). The fetch unit in EXACT-S features two special, software-managed registers, *base register* and *offset register*, which enable the fetch unit to calculate the addresses of array elements as the branches that test them are fetched. This way, a branch's ID is known when the branch is fetched, so that the explicit predictor can be indexed directly by the current branch ID.

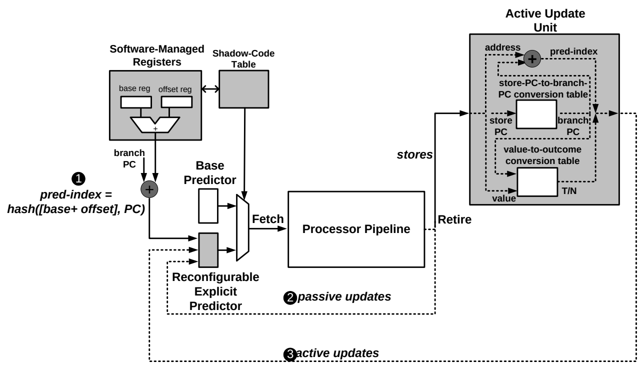


Figure 2. High-level view of EXACT-S.

A light-weight run-time layer allocates the base register and offset register for the static branch that tests elements of an array. Software writes the base address of the array into the base register, increments/decrements the offset register, and signals when to index the explicit predictor for a dynamic instance of the static branch. The index is the sum of the base and offset registers, hashed with the branch's PC to form the ID, as shown in Figure 2.

One approach to enable software to manage the fetch unit is to modify the instruction-set architecture (ISA): (1) add new instructions to write the base and offset registers, and (2) add a bit to conditional branch opcodes to signal whether a static branch should use the explicit predictor or default predictor. There are two drawbacks to this approach. First, it requires changing the ISA which may not be an option. Second, this approach will increase the dynamic instruction count of programs modified to use EXACT-S, reducing the performance gain of EXACT-S.

Therefore, we propose managing the fetch unit with *shadow code*. We use the term "shadow" because the light-weight run-time layer creates a correspondence between selected instructions in the original program and shadow-instructions in the shadow code. That is, each shadow-instruction *shadows* a particular instruction in the original program. Each shadow-instruction is tagged with the PC of the instruction that it shadows. When the instruction is fetched, the shadow-instruction that shadows it is triggered via the PC.

Typically, key instructions that are shadowed by shadow-instructions include: (1) the instruction that generates the base address of an array, (2) any instruction that is convenient for signaling when to increment the offset and by what positive or negative value, and (3) the static branch that tests elements of the array. For example, consider a loop that iterates over an array. The instruction prior to the loop that generates the base address of the array is shadowed by a shadow-instruction in the shadow code called the seed shadow-instruction. The seed shadow-instruction writes the base address into the base register in the fetch unit. An arbitrary instruction prior to the loop is shadowed by a shadow-instruction to initialize the offset register. A convenient instruction within the loop is shadowed by a shadow-instruction that increments/decrements the offset register by a certain stride. Finally, a branch within the loop that tests elements of the array is shadowed by a shadow-instruction which signals that the branch should be predicted with the explicit predictor. The load address feeding the branch is computed using the base and offset registers.

The seed shadow-instruction is special in that it involves communication between the general-purpose register file and the fetch unit's base register (to write a value into the base register). The seed shadow-instruction records the physical register number allocated to the destination of the instruction that it shadows. This way, when the instruction executes, its result (the base address) can be obtained from the function unit's result bus. (Note that the hardware support for this is present in most modern processors as the datapath for sending indirect-branch targets from the execution engine to the fetch unit.) Moreover, there is a ready bit associated with the base register in the fetch unit. The ready bit is reset when the seed shadow-instruction is initially triggered and set when the shadowed instruction executes. If a branch is fetched that needs to index the explicit predictor, but the base register is not yet ready, then the branch is predicted by the default predictor.

In addition to the accuracy benefits that come with software-controlled indexing, a number of complex and expensive mechanisms are made obsolete:

1. Note that EXACT-S does not require a chooser (contrast Figure 1 with Figure 2). The shadow code, and the readiness of the base register, controls selection of the explicit predictor or default predictor.

2. EXACT-S does not require the complex post-retirement ID generation unit for propagating loads' addresses to their dependent branches. IDs generated in the fetch unit obviate the need for this complex hardware.

3. Since the current branch's ID is used to index into the explicit predictor, there is no need for an address-to-index conversion table in the active update unit. As shown in Figure 2, the active update unit simply hashes the store address with the branch PC to determine the index of the branch that must be updated. Only the small store-PC-to-branch-PC and value-to-outcome tables remain. As an alternative to using hardware to train these tables, they are pre-loaded with information generated by the light-weight run-time layer.

While we focused on arrays in the description above, it is possible to target EXACT-S for stable linked-lists as well, as we discuss in Section 3. Essentially, they can be treated as logical arrays in a process called "array-ification".

3. EXACT-S IMPLEMENTATION

3.1 Shadow Code and the Shadow-Code Table

Shadow code is conveyed from software to hardware without changing the ISA, as follows:

- The shadow code is made a part of the data segment of the program binary. This allows for user-level loads to read bytes of the shadow code.
- The Shadow-Code Table is memory-mapped, making it possible to initialize its contents via user-level stores. Typically, a given machine comes with a machine manual that specifies its memory-mapped registers.
- Initialization code is added to the beginning of the program, that uses pairs of loads and stores to copy the bytes of shadow code from the data segment (loads) to the Shadow-Code Table (stores).
- If the run-time layer would like to use more shadow-instructions than there is space in the Shadow-Code Table, the table can be explicitly managed even after the initialization stage. Bytes of shadow code can be loaded from the data segment and stored into the Shadow-Code Table as different shadow-instructions become needed.

There is a one-to-one correspondence between selected instructions in the program and shadow-instructions in the shadow code. That is, each shadow-instruction *shadows* a particular instruction in the program. Each shadow-instruction has a PC field containing the PC of the instruction that it shadows. When the instruction is fetched, the shadow-instruction that shadows it is triggered via the PC. This process is explained next.

In parallel with fetching instructions from the instruction cache, the PCs of all instructions in the fetch bundle are searched in the Shadow-Code Table. If a PC hits, then the matching shadow-instruction is read from the Shadow-Code Table and executed within the fetch unit. Shadow-instructions write, update, or read the software-managed registers in the fetch unit.

If a branch hits in the Shadow-Code Table, it will be predicted by the explicit predictor instead of the default predictor. This means that the Shadow-Code Table assumes the role of the chooser as shown in Figure 2.

3.2 Software-Managed Registers in the Fetch Unit

Figure 3 shows the fields of the two software-managed registers in the fetch unit read and written by the shadow code.

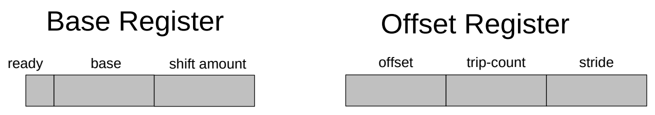


Figure 3. The fetch unit registers read and written by the shadow code.

3.2.1 Base Register

The base register is used for conveying the base address of a data structure that is being traversed, such as an array or linked-list, to the fetch unit. There are two fields in a base register: *base* and *shift amount*. *Base* contains the base address. *Shift amount* is used by software for effectively modeling linked-lists as arrays in the explicit predictor, a process we refer to as "array-ifying" the linked-list. Array-ification is discussed below. Figure 3 also shows a ready bit associated with the base register. The ready bit is not visible to software, *i.e.*, it is purely microarchitectural. It will be explained when we discuss the shadow-instruction type that writes base registers.

To array-ify a linked-list, the address of its head node is considered the base address and subsequent nodes are considered to be at contiguous addresses with a stride of 1 (even though they are at arbitrary addresses). Thus, when sequencing a linked-list, dynamic branches that test its elements index a contiguous range of entries in the explicit predictor.

Basic array-ification, as described above, may cause different linked-lists to conflict in the explicit predictor because it is possible for their contiguous ranges to overlap. Whether or not two array-ified linked-lists conflict in the explicit predictor depends on their base addresses and sizes. True arrays cannot conflict in this way because they are laid out contiguously in memory; true arrays can only conflict in the explicit predictor due to its limited size.

The *shift amount* field is used by software to reduce conflicts between array-ified linked-lists. The *shift amount* field specifies an amount by which the base address should be shifted to the left when forming an index into the explicit predictor. Effectively, this provides a dedicated region for the linked-list within the explicit predictor as long as the number of elements is less than $2^{(\text{shift amount})}$.

3.2.2 Offset Register

The fields of the offset register are designed with loops in mind, in particular, loops that iterate over and test the contents of data structures such as arrays and linked-lists. An offset register has three fields: *offset*, *trip-count*, and *stride*. The *offset* field generally corresponds to the loop induction variable. The *trip-count* is the number of times to iterate. The *stride* is the stride between elements that are accessed consecutively. It is a signed integer.

Thus, the offset register has two uses. First, the address of an element being tested can be calculated by adding *offset*stride* to the base address contained in a base register. Second, *offset* can be compared to *trip-count* to predict the loop branch: taken if they are not equal and not-taken if they are equal.

3.3 Shadow-Instructions

Figure 4 shows the format of a shadow-instruction. The *PC* field identifies the program instruction that is shadowed. The *op-code* field is 3-bits wide, supporting up to 8 different types of shadow-instructions. The last field is *immediate/flag*. This field allows the shadow-instruction to use an immediate value. Alternatively, some shadow-instructions use this field as a flag that controls incrementing the offset register.

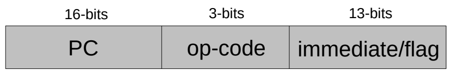


Figure 4. General format of a shadow-instruction.

The shadow-instructions used in this paper are enumerated below:

3.3.1 seed shadow-instruction

The *seed* shadow-instruction shadows an instruction in the program that generates base addresses of arrays or linked-lists. The fields relevant to the *seed* shadow-instruction are: *PC* and *op-code*

The *seed* shadow-instruction is the only one that cannot be executed immediately in the fetch unit because it must wait for its corresponding program instruction to execute. The *seed* shadow-instruction is executed as follows. First, the ready bit of the base register is cleared since the base address is not yet available. (Branches that are directed to use the explicit predictor – see *regular-branch* shadow-instruction – must use the default predictor if they reference a not-ready base register.) At the same time, its corresponding program instruction is annotated so that it knows to communicate with the fetch unit when it reaches two different pipeline stages: the rename stage and the writeback stage. When the program instruction is renamed, it sends the physical register tag of its destination register to the fetch unit. The fetch unit writes the physical register tag into the base register, in lieu of an actual value. When the program instruction has executed and reaches the writeback stage, it communicates its physical register tag and value to the fetch unit. The fetch unit writes the value into the base register and sets the ready bit, *only* if the physical register tag matches the one currently in the base register (otherwise the value is simply discarded). The reason for comparing physical register tags is to handle the scenario of a second *seed* shadow-instruction reusing the base register before the first *seed* shadow-instruction has had a chance to write it. At this point, branches affiliated with the first *seed* shadow-instruction have already been predicted without the benefit of using the explicit predictor, moreover, the first *seed* shadow-instruction should not interfere with the branches affiliated with the second *seed* shadow-instruction that is now in progress. This implementation is basically an application of Tomasulo's renaming algorithm [2].

3.3.2 init-offset shadow-instruction

This shadow-instruction initializes all three fields of the offset register. It initializes the *offset* field to zero and the *trip-count* and *stride* fields to specified values. Aside from *PC* and *op-code*, the field relevant to the *init-offset* shadow-instruction is: *immediate/flag* that specifies the *trip-count* and *stride* values.

3.3.3 loop-branch shadow-instruction

This shadow-instruction signals that the corresponding program instruction is a loop branch, and it will generate a prediction by comparing the *offset* and *trip-count* fields of the offset register. If they match, it means that the loop branch will fall-through and the prediction should be not-taken, otherwise, the prediction should be taken. Executing this shadow-instruction will also automatically increment the *offset* field of the offset register. The fields relevant to the *loop-branch* shadow-instruction are: *PC* and *op-code.*

3.3.4 regular-branch shadow-instruction

This shadow-instruction signals that the corresponding program instruction is a branch, that should attempt to use the explicit predictor. The index into the explicit predictor is calculated by adding the base register (shifted for array-ified linked-lists) to the offset register and combining with the PC, as follows:

$$\text{hash}\{\text{PC}, ((\textit{base} << \textit{shift amount}) + (\textit{offset} * \textit{stride}))\}$$

The multiplication of the *offset* and *stride* can be performed by a shift operation if the stride is constrained to be a power of two. Alternatively, hardware can circumvent all three arithmetic operations – the shift, multiply, and add – by microarchitecturally extending the base/offset registers with an absolute address register that is initialized to (*base<<shift amount*) and incremented by *stride*.

Aside from *PC* and *op-code*, the field relevant to the *regular-branch* shadow-instruction is: *immediate/flag* (if 1, increment *offset*).

3.3.5 linked-list-store shadow-instruction

After array-ification, the index of a linked-list element is not based on its address. Instead, it is based on the address of the head element plus a virtual offset. This complicates active updates for linked-list elements: the index cannot be inferred from the store address. The software solution to this problem is to exploit opportunities to perform the store in the context of a linked-list traversal. If done in the context of a traversal, the store can infer its array-ified address the same way a branch does. This is achieved using the *linked-list-store* shadow-instruction. This shadow-instruction signals that the corresponding program instruction is a store that should calculate its active-update index using the base register and offset register, rather than use its own address for the active-update index. The partial index (does not include branch PC yet) is calculated in the fetch stage but will not be used until the store retires. Thus, the partial index is sent with the store down the pipeline until the dispatch stage at which time the partial index can be placed in the store's store buffer entry for use at retirement.

Note, it is not a requirement that linked-list stores be performed in the context of a linked-list traversal. If it cannot be done in a linked-list traversal, there are other options depending on the situation:

- If the run-time layer knows that the store is rare or that it rarely flips branch outcomes, then it may be deemed unnecessary to trigger active updates by the store. Active updates by a particular store PC can be disabled simply by excluding it from the store-PC-to-branch-PC table in the active update unit.
- If the run-time layer believes that the store frequently flips branch outcomes, then the affected static branches can be relegated to the default predictor, simply by not shadowing them with *regular-branch* shadow-instructions.

The fields relevant to the *linked-list-store* shadow-instruction are: *PC* and *op-code.*

3.3.6 *Summary of shadow-instructions*

Table 1 shows a summary of the shadow-instructions used in this paper. The table shows, for each shadow-instruction, the *op-code* and *immediate/flag* fields, as well as the operations performed by the shadow-instruction.

3.4 Light-weight Run-time Layer

The run-time layer can be a simple software layer that monitors loop regions in which lots of mispredictions happen. The run-time layer inspects the highly mispredicted branches, in the monitored loop, to see if the branch is testing a value loaded from an array (or a linked-list). If so, the run-time layer will generate the shadow code corresponding to this loop. This paper does not discuss the details of the light-weight run-time layer, but focuses on its usage to generate the shadow code and to support the EXACT-S predictor.

3.5 Active Update Unit

The active update unit consists of the store-PC-to-branch-PC and value-to-outcome conversion tables (Figure 2).

In EXACT, store-PC-to-branch-PC conversion was wrapped into the address-to-index conversion table [1]. Since EXACT-S eliminates the address-to-index conversion table, it has a dedicated store-PC-to-branch-PC conversion table. It is a small content-addressable table accessed by store PC and a match provides the affected branch PC. The branch PC and store address are hashed to determine the index to actively update.

For value-to-outcome conversion, EXACT used two different reuse tables, a Ranges Reuse Table (RRT) and a General Reuse Table (GRT) [1]. EXACT-S borrows only the RRT. It is a small content-addressable table accessed by branch PC and a match provides two value ranges: the first provides a range of store values for which the prediction should be updated to taken and the second provides a range of store values for which the prediction should be updated to not-taken.

While it is possible to dynamically train the store-PC-to-branch-PC table and the RRT, it is much more efficient to populate them via software (reduces hardware overhead, design complexity, and training time and power), in the same way that the Shadow-Code Table is populated: their contents are part of the data segment and they are populated using pairs of loads and memory-mapped stores in the initialization phase of the program.

3.6 Reconfigurable Explicit Predictor

A practical consideration for the explicit predictor is that it should be possible to reallocate its storage to the default predictor, as there will be programs that do not exploit EXACT-S for various reasons: (1) legacy programs, (2) programs that have high accuracy with history-based branch predictors, or (3) programs that have branches not suitable for EXACT-S.

Figure 5 shows our reconfigurable predictor based on the aggressive history-based L-TAGE predictor [14]. L-TAGE is composed of a bimodal predictor, a loop predictor, and 12 partially-tagged predictors (T1 through T12) that use different lengths of folded global branch history based on a geometric series and ranging from 4 bits to 640 bits. When EXACT-S is enabled, the explicit predictor consists of four banks on the right labeled T3b through T6b. When EXACT-S is disabled, L-TAGE uses these four banks in lieu of its tables T3a through T6a. We allocate roughly half of the total storage to each of the default predictor and the explicit predictor. This implies that T3b-T6b are much larger than their counterparts T3a-T6a, hence, it is not too wasteful to not use T3a-T6a when EXACT-S is disabled. In Section 5, we observe that when the reconfigurable predictor is configured for L-TAGE-only operation (EXACT-S disabled), it performs about the same as a fixed L-TAGE predictor of the same size.

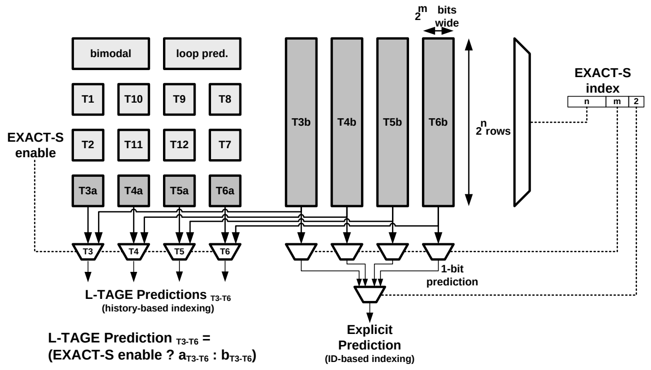


Figure 5. Reconfigurable predictor based on L-TAGE.

When EXACT-S is enabled, the four banks T3b-T6b are used as

Table 1. Summary of shadow-instructions used in this paper.

shadow-instruction	op-code	immediate/ flag	operations
seed	0x0	shift amount	1. base-register[*base*] = GPR_X * use base-register[*ready*] to synchronize with shadowed instruction X (see relevant text) 2. base-register[*shift-amount*] = immediate
init-offset	0x1	trip-count and stride	1. offset-register[*offset*] = 0 2. offset-register[*trip-count*] = immediate_upper 3. offset-register[*stride*] = immediate_lower
loop-branch	0x4	N/A	1. outcome = (offset-register[*offset*] == offset-register[*trip-count*] ? NT : T) 2. offset-register[*offset*]++
regular-branch	0x6	increment offset	1. pred-index = hash(branch-PC, ((base-register[*base*]<<base-register[*shift amount*]) + (offset-register[*offset*]*offset-register[*stride*]))) 2. if (flag) offset-register[*offset*]++
linked-list-store	0x7	N/A	active-update-address = ((base-register[*base*]<<base-register[*shift amount*]) + (offset-register[*offset*]*offset-register[*stride*]))

one logical predictor that outputs a 1-bit prediction. This is achieved by dividing the EXACT-S index into three parts: row selection, column selection within each bank, and bank selection.

Implementing a reconfigurable gshare predictor [11] would be easier as it would require just two tables that are configured as either gshare+explicit (1 table each) or gshare-only (2 tables operating as 1).

4. SHADOW CODE EXAMPLE AND OTHER POTENTIAL USE-CASES

Figure 8 (in the appendix) shows the source code, assembly code, and shadow code generated for two loops in gzip. The two loops are similar in behavior, so we will explain only the shadow code of the first loop. The branches of interest are highlighted in red in the assembly. Assembly instructions that are shadowed are highlighted in bold and connected via arrows to their shadow-instructions. In each loop, the code sequences through a 32K-entry array, including continuously modifying its elements (thus, active updates are crucial). The instruction at address 0x4013d8 is the instruction that will generate the base address of the first array, so it is shadowed by a seed shadow-instruction that writes the base register (base = r7, the destination of the shadowed instruction, and shift amount = 0). The instruction at address 0x4013e0 is shadowed, to initialize the offset (to 0), trip-count (to 32,768), and stride (to 2) fields of the offset register. The instruction at address 0x401480 is the loop branch which will be shadowed by a loop-branch shadow-instruction (tests and increments the offset register). And finally, the instruction at address 0x401440 is the branch of interest, which is shadowed by a regular-branch shadow-instruction.

In addition to the branch prediction use-case presented in this paper, shadow-instructions – *i.e.*, direct microarchitecture manipulation instructions – can be used for other purposes, including and not limited to:

- *Fetch gating*: Shadow-instructions can direct the fetch unit to stop fetching upon encountering hard-to-predict branches that tend to depend on expensive cache misses.
- *Resource configuration*: Upon encountering very serial regions, shadow-instructions can (a) direct the fetch unit to not fetch at full bandwidth or (b) direct the processor to transition to a lower-power mode.

5. RESULTS

5.1 Methodology

All results are based on a custom, detailed cycle-level processor simulator derived from the SimpleScalar toolset [4]. Parameters of the modeled processor are shown in Table 2.

Table 2. Microarchitecture configuration.

L1 I&D Caches	64KB, 4-way, 64B line, hit=1 cycle, miss=10 cycles, 32 MHSRs
L2 Cache	Unified, 2MB, 8-way, 128B line, 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

Eleven of the SPEC2K integer 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 [16] was used to locate representative simulation points.

We used the source code posted by Seznec [15] for the L-TAGE predictor. We performed a design space exploration to find good

Table 3. Predictor configurations used in the experiments.

Predictor	Access Latency (cycles)	Fixed-Size Components (KB)	Reconfigurable-EX (KB)	Bimodal (KB)	Loop (KB)	Tagged (KB)											
						1	2	3a	4a	5a	6a	7	8	9	10	11	12
3.3 KB L-TAGE	3 cycles	0	0	2.50	0.41	0.02	0.02	0.05	0.05	0.05	0.06	0.03	0.03	0.03	0.04	0.02	0.02
19.9 KB EXACT-S	4 cycles	0.57	16	2.50	0.41	0.02	0.02	0.05	0.05	0.05	0.06	0.03	0.03	0.03	0.04	0.02	0.02
16.7 KB L-TAGE	4 cycles	0	0	2.50	0.41	0.75	0.75	1.62	1.62	1.75	1.88	1.00	1.06	1.06	1.12	0.59	0.62
49.3 KB EXACT-S	5 cycles	0.57	32	2.50	0.41	0.75	0.75	1.62	1.62	1.75	1.88	1.00	1.06	1.06	1.12	0.59	0.62
58.3 KB L-TAGE	5 cycles	0	0	2.50	0.41	3.00	3.00	6.50	6.50	7.00	7.50	4.00	4.25	4.25	4.50	2.38	2.50

Table 4. Fixed-size subcomponents of EXACT-S (not including default predictor and explicit predictor).

Unit	Structure	# of entries / organization	Contents per entry	Size (KB)
Fetch Unit	Shadow-Code Table	64 entries / fully-assoc.	16-bit PC tag + 16-bit shadow-instruction payload	0.24
	Software-Managed Registers	2 registers	1 ready bit + 20-bit base + 5-bit shift-amount + 13-bit offset + 13-bit trip-count + 3-bit stride	0.007
Active Update Unit	RRT	8 entries / 4-way set-assoc.	1 valid bit + 2-bit LRU + 13-bit tag + 4x16-bit for (min_NT, max_NT, min_T, max_T)	0.08
	Store-PC-to-Branch-PC Conversion Table	16 entries / 4-way set-assoc.	1 valid bit + 2-bit LRU + 12-bit tag + 8x14-bit branch-PCs	0.24
Total cost of fixed subcomponents (does not include default predictor and explicit predictor):				0.57

fixed sizes for its bimodal and loop predictors. The sizes of the tagged components are based on equations in the L-TAGE source code. Table 3 shows the three L-TAGE configurations we use in the experiments that follow, including total size and the sizes of their sub-components.

These three L-TAGE configurations will be used both standalone and in the context of our reconfigurable explicit predictor (notice we use the same notation from Section 3.6 for tagged tables 3a through 6a). We evaluated two sizes for the reconfigurable explicit predictor subcomponent, 16KB and 32KB (also in Table 3). The 16KB reconfigurable explicit predictor is integrated with the 3.3KB L-TAGE and the 32KB one is integrated with the 16.7 KB L-TAGE. We evaluate the five configurations under two different latency assumptions: 1-cycle latency for all configurations and N-cycle latency based on total size as shown in Table 3. For the N-cycle case, all configurations are used as overriding predictors [10] in conjunction with a 1-cycle 8KB gshare predictor.

Table 4 shows the cost of the fixed-size subcomponents of EXACT-S. They total to about ½ KB. This cost does not include the cost of the reconfigurable explicit predictor's subcomponents: the default and explicit predictors.

5.2 Results

Figure 6 shows the misprediction rates of the five configurations. For the benchmarks that do not use ID-based indexing, 19.9 KB EXACT-S and 49.3 KB EXACT-S achieve similar accuracies as the 16.7 KB and 58.3 KB L-TAGE predictors, respectively. As expected, on benchmarks that use ID-based indexing, 19.9 KB EXACT-S outperforms the 16.7 KB L-TAGE for gzip (6% vs. 7%) and twolf (3.5% vs. 4%). Similarly, 49.3 KB EXACT-S outperforms the 58.3 KB L-TAGE for gzip (4.7% vs. 7.2%) and twolf (2.1% vs. 3.9%). In summary, all benchmarks across the board have benefited from the reconfigurable predictor compared to an equally sized fixed L-TAGE predictor.

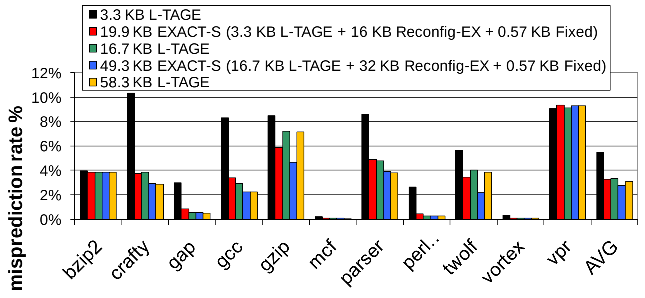
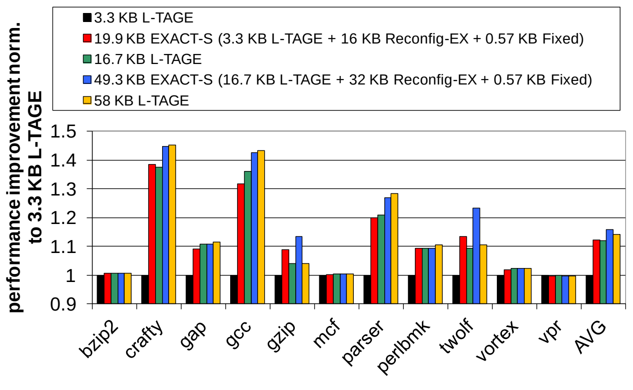


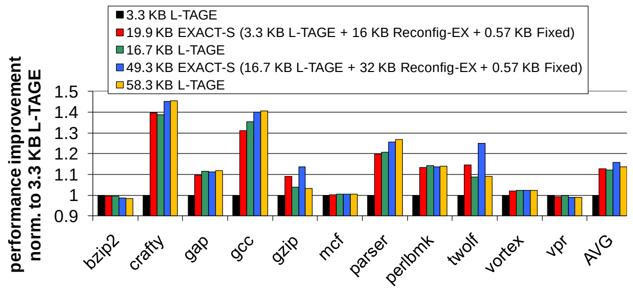
Figure 6. Misprediction rates.

Figure 7 shows performance improvement normalized to 3.3KB L-TAGE assuming (a) 1-cycle latency and (b) N-cycle latency overriding predictor based on values in Table 3. 19.9 KB EXACT-S achieves similar performance compared to 16.7 KB L-TAGE for all non-applicable benchmarks, except for gcc where minor slowdown is recorded, and speedups of 5% for gzip and twolf. Similarly, 49.3 KB EXACT-S and 58.3 KB L-TAGE achieve similar performance on all non-applicable benchmarks, while 49.3 KB EXACT-S shows speedups of 10% and 15% for gzip and twolf, respectively, compared to 58.3 KB L-TAGE.

Table 5 compares mispredictions rates for EXACT [1] and EXACT-S at comparable cost points. For EXACT, we show two



(a) 1-cycle latency



(b) N-cycle latency, overriding predictors

Figure 7. Performance improvement w.r.t. 3.3 KB L-TAGE.

different flavors, depending on whether the SACT (the address-to-index conversion table used for active updates) is implemented in dedicated storage versus virtualized. We only show comparisons for gzip and twolf. EXACT-S significantly outperforms EXACT at these cost points. EXACT-S is more accurate for equal or lesser cost because of its direct-indexing strategy and very low cost active updates. In the EXACT paper [1], the authors show that EXACT can deliver higher accuracies at expensive cost points. Even at these points, EXACT underperforms EXACT-S due to indexing with a prior branch's ID instead of the current one. Moreovoer, for the rest of the benchmarks, EXACT is a major liability as its resources are not reconfigured to benefit the base predictor (L-TAGE), while EXACT-S will deliver similar results to an equally-sized fixed L-TAGE predictor (results in Figure 7).

Table 5. Comparing EXACT and EXACT-S.

	EXACT, dedicated SACT		**EXACT. virtualized SACT**		**EXACT-S**	
	size (KB)	**misp. rate**	**size (KB)**	**misp. rate**	**size (KB)**	**misp. rate**
gzip	62	7.33 %	68	6.48 %	49.3	4.71%
twolf	67	7.30 %	68	7.03 %	49.3	2.11%

6. RELATED WORK

With the advent of two-level adaptive branch prediction [18], 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 target the load-branch idiom or use software management of hardware.

EXACT-S borrows principles from EXACT [1] and, through software intervention, yields a significantly simpler application of these principles. EXACT was discussed at length in Section 2.1.

The ARVI predictor [6] uses live-in register values of a 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 generating load values or addresses early. EXACT-S exploits software intervention to achieve the latter.

The ABC predictor (address-branch correlation) [9] specifically targets 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. EXACT-S (and its precursor EXACT) has the more formidable challenge of hiding the core pipeline latency for all branches, requiring the load addresses for every branch to be available. EXACT-S exploits software intevention to make this possible and practical.

The base and offset registers of EXACT-S are reminiscent of conventional stride predictors used for load address prediction [13] and data prefetching. We considered hardware address prediction but did not pursue it because it breaks down when the base address is highly variable. For example, twolf's array-traversing loop traverses many different array objects. The most important facet of EXACT-S is in providing the base address early; strided access is otherwise a fairly common idiom to be exploited.

Farcy et al. [8], Roth and Sohi [12], and Zilles and Sohi [19] proposed extracting, hoisting, and pre-executing the backward slices of hard-to-predict branches so that their outcomes are known by the time they are fetched.

In SSMT [5], the authors suggested using a micro-thread to manage a large PAg branch predictor [18] stored in main memory. After each branch is fetched, a micro-thread is spawned to update the branch predictor and prepare a prediction for the next branch instance, which is stored in a prediction cache.

In DISE [7], the authors suggested using a pattern table to match on a sequence of instructions and replace them with either a native sequence of instructions or micro-instructions which are fetched from a replacement table. They applied DISE to memory fault isolation and dynamic code compression. Our proposed shadow-instructions do not replace program instructions but enable them to transparently trigger actions in the fetch unit.

The IBM POWER ISA features a branch-on-count instruction which manages loop counter registers in the branch unit.

7. SUMMARY AND FUTURE WORK

This paper presented a novel software-managed reconfigurable branch predictor, EXACT-S, that accurately predicts load-dependent branches that sequence large data structures. In EXACT-S, software conveys key information directly to the fetch unit that it can use to generate branches' load addresses in a timely manner which in turn is essential for providing them with dedicated predictions. This is the same principle behind the precursor EXACT predictor, but EXACT-S is significantly streamlined in comparison. We demonstrated EXACT-S on two applications, gzip and twolf, by writing shadow code for some of their most difficult-to-predict branches. EXACT-S removes 33% of mispredictions in gzip and 50% of mispredictions in twolf, compared to a similarly-sized aggressive history-based L-TAGE predictor. For other applications, the unused explicit predictor storage is absorbed by the default predictor via a reconfigurable design: the reconfigured predictor shows similar accuracy to a similarly-sized fixed L-TAGE predictor.

For future work, we plan to expand the repertoire of sequencing idioms of EXACT-S. We also plan to explore other performance and energy optimizations enabled by direct manipulation of processor units by shadow code.

8. ACKNOWLEDGEMENTS

This research was supported in part 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 represent the views of the National Science Foundation.

9. REFERENCES

[1] M. Al-Otoom, E. Forbes, E. Rotenberg. EXACT: Explicit Dynamic-Branch Prediction with Active Updates. *CF-7*, May 2010.

[2] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo. The IBM System/360 Model 91. *IBM Journal of R&D*, vol. 11, 1967.

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

[4] D. Burger, et al. Evaluating Future Microprocessors: The Simplescalar Toolset. CS-TR-96-1308,UW-Mad., July 1996.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, Y. Patt. Simultaneous Subordinate Microthreading (SSMT). *ISCA-26*, May 1999.

[6] L. Chen, S. Dropsho, D. Albonesi. Dynamic Data Dependence Tracking and its Application to Branch Prediction. *HPCA-9*, Feb. 2003.

[7] M. L. Corliss, E. C. Lewis, A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. *ISCA-30*, June 2003.

[8] A. Farcy, O. Temam, R. Espasa, T. Juan. Dataflow Analysis of Branch Mispredictions and its Applications to Early Resolution of Branches. *MICRO-31*, Dec. 1998.

[9] H. Gao, Y. Ma, M. Dimitrov, H. Zhou. Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches. *HPCA-14*, Feb. 2008.

[10] D. Jiménez, S. Keckler, C. Lin. The Impact of Delay on the Design of Branch Predictors. MICRO-33, Dec. 2000.

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

[12] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. *HPCA-7*, Jan. 2001.

[13] Y. Sazeides, J. E. Smith. The predictability of data values. *MICRO-30*, Dec. 1997.

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

[15] A. Seznec. L-TAGE header file. http://www.irisa.fr/caps/projects/Architecture/L-TAGE.h

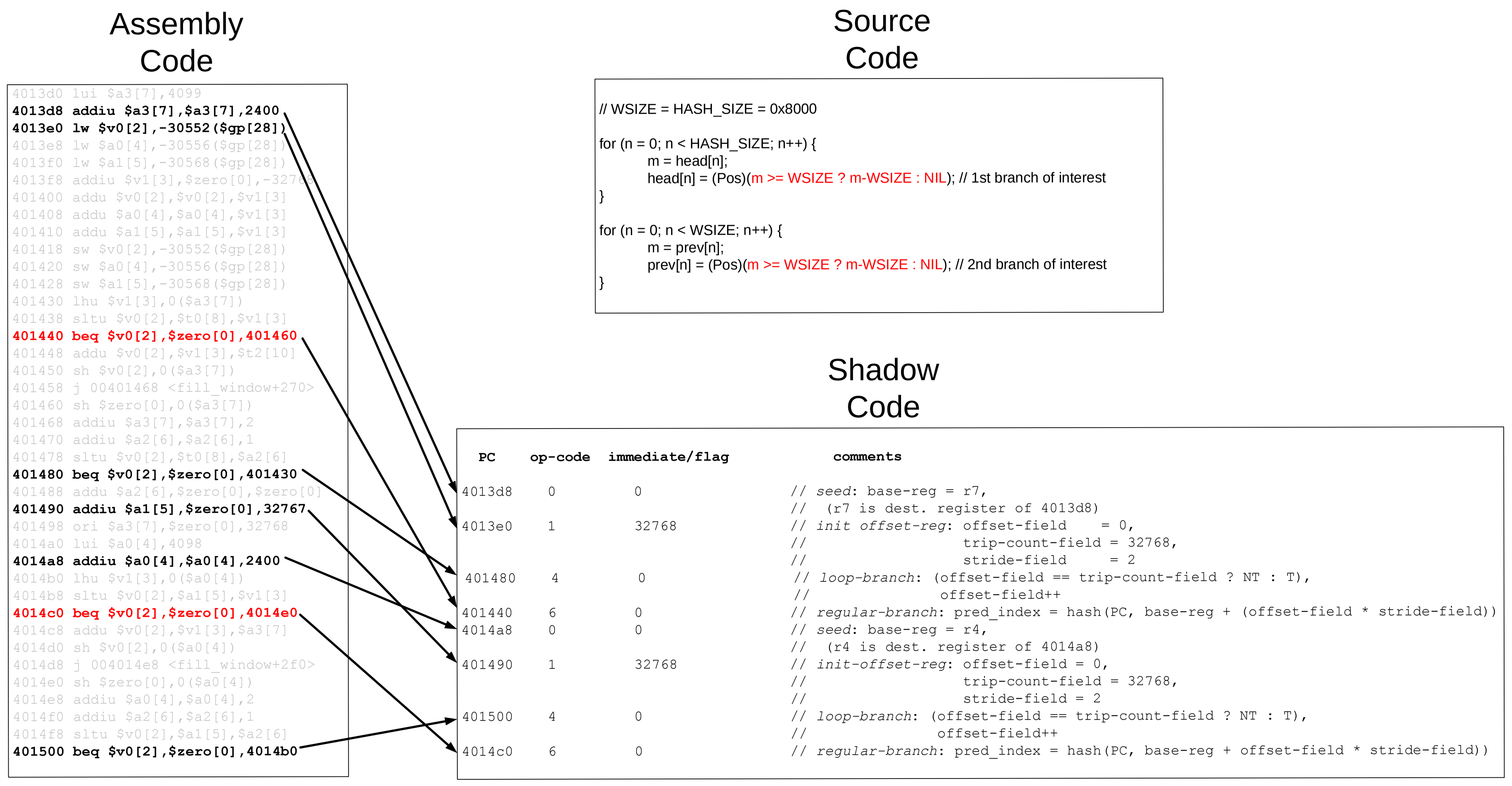
[16] T. Sherwood, et al. Automatically Characterizing Large Scale Program Behavior. *ASPLOS-X*, Oct. 2002.

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

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

[19] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. *ISCA-28*, July 2001.

10. APPENDIX



```
4013d0 lui $a3[7],4099
4013d8 addiu $a3[7],$a3[7],2400
4013e0 lw $v0[2],-30552($gp[28])
4013e8 lw $a0[4],-30556($gp[28])
4013f0 lw $a1[5],-30568($gp[28])
4013f8 addiu $v1[3],$zero[0],-32768
401400 addu $v0[2],$v0[2],$v1[3]
401408 addu $a0[4],$a0[4],$v1[3]
401410 addu $a1[5],$a1[5],$v1[3]
401418 sw $v0[2],-30552($gp[28])
401420 sw $a0[4],-30556($gp[28])
401428 sw $a1[5],-30568($gp[28])
401430 lhu $v1[3],0($a3[7])
401438 sltu $v0[2],$t0[8],$v1[3]
401440 beq $v0[2],$zero[0],401460
401448 addu $v0[2],$v1[3],$t2[10]
401450 sh $v0[2],0($a3[7])
401458 j 00401468 <fill_window+270>
401460 sh $zero[0],0($a3[7])
401468 addiu $a3[7],$a3[7],2
401470 addiu $a2[6],$a2[6],1
401478 sltu $v0[2],$t0[8],$a2[6]
401480 beq $v0[2],$zero[0],401430
401488 addu $a2[6],$zero[0],$zero[0]
401490 addiu $a1[5],$zero[0],32767
401498 ori $a3[7],$zero[0],32768
4014a0 lui $a0[4],4098
4014a8 addiu $a0[4],$a0[4],2400
4014b0 lhu $v1[3],0($a0[4])
4014b8 sltu $v0[2],$a1[5],$v1[3]
4014c0 beq $v0[2],$zero[0],4014e0
4014c8 addu $v0[2],$v1[3],$a3[7]
4014d0 sh $v0[2],0($a0[4])
4014d8 j 004014e8 <fill_window+2f0>
4014e0 sh $zero[0],0($a0[4])
4014e8 addiu $a0[4],$a0[4],2
4014f0 addiu $a2[6],$a2[6],1
4014f8 sltu $v0[2],$a1[5],$a2[6]
401500 beq $v0[2],$zero[0],4014b0
```

```
// WSIZE = HASH_SIZE = 0x8000

for (n = 0; n < HASH_SIZE; n++) {
        m = head[n];
        head[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL); // 1st branch of interest
}

for (n = 0; n < WSIZE; n++) {
        m = prev[n];
        prev[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL); // 2nd branch of interest
}
```

```
PC      op-code  immediate/flag           comments

4013d8    0         0              // seed: base-reg = r7,
                                   //  (r7 is dest. register of 4013d8)
4013e0    1         32768          // init offset-reg: offset-field    = 0,
                                   //                  trip-count-field = 32768,
                                   //                  stride-field     = 2
401480    4         0              // loop-branch: (offset-field == trip-count-field ? NT : T),
                                   //              offset-field++
401440    6         0              // regular-branch: pred_index = hash(PC, base-reg + (offset-field * stride-field))
4014a8    0         0              // seed: base-reg = r4,
                                   //  (r4 is dest. register of 4014a8)
401490    1         32768          // init-offset-reg: offset-field = 0,
                                   //                  trip-count-field = 32768,
                                   //                  stride-field = 2
401500    4         0              // loop-branch: (offset-field == trip-count-field ? NT : T),
                                   //              offset-field++
4014c0    6         0              // regular-branch: pred_index = hash(PC, base-reg + offset-field * stride-field))
```

Figure 8. Shadow-code example for gzip.