Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches

Hongliang Gao Yi Ma Martin Dimitrov Huiyang Zhou
School of Electrical Engineering and Computer Science, University of Central Florida
{hgao, yma, dimitrov, zhou}@cs.ucf.edu

Abstract

Hard-to-predict branches depending on long-latency cache-misses have been recognized as a major performance obstacle for modern microprocessors. With the widening speed gap between memory and microprocessors, such long-latency branch mispredictions also waste substantial power/energy in executing instructions on wrong paths, especially for large instruction window processors.

This paper presents a novel program locality that can be exploited to handle long-latency hard-to-predict branches. The locality is a result of an interesting program execution behavior: for some applications, major data structures or key components of the data structures tend to remain stable for a long time. If a hard-to-predict branch depends on such stable data, the address of the data rather than the data value is sufficient to determine the branch outcome. This way, a misprediction can be resolved much more promptly when the data access results in a long-latency cache miss. We call such locality address-branch correlation and we show that certain memory-intensive benchmarks, especially those with heavy pointer chasing, exhibit this locality. We then propose a low-cost auxiliary branch predictor to exploit address-branch correlation. Our experimental results show that the proposed scheme reduces the execution time by 6.3% (up to 27%) and energy consumption by 5.2% (up to 24%) for a set of memory-intensive benchmarks with a 9kB prediction table when used with a state-of-art 16kB TAGE predictor.

1. Introduction

The performance of modern microprocessors for single-threaded applications is mainly constrained by two factors, long-latency memory accesses and hard-to-predict branches. Although large instruction-window processors [1],[5],[8],[17],[20] can effectively exploit memory-level parallelism to overcome the memory wall problem, the pressure is essentially shifted to branch predictors to fetch a large number (thousands) of instructions from correct paths. Particularly, if a program features hard-to-predict branches whose operands depend on long-latency cache-misses, those high-penalty mispredictions become serious performance obstacles and cause substantial energy to be wasted in executing instructions from wrong paths.

It has been shown in the 2nd JILP Championship Branch Prediction Competition (CBP-2) [21] that scaling the state-of-art branch correlation based branch predictor unlimitedly can only reduce the misprediction rates by about 21% compared to a 32kB realistic predictor [15]. To further improve prediction accuracy, we have to discover novel locality that can be explored to predict those branches, which are not predicted accurately by exploring branch correlation. Pre-execution has been proposed as one solution to handle hard-to-predict branches [4],[12]. Besides the additional cost and complexity, pre-execution is less effective for large instruction window processors since it is difficult for a pre-execution thread to be far ahead of a main thread with a large instruction window.

We focus on hard-to-predict branches that depend on long-latency cache-missing loads and we refer to them as *hard-to-predict long-latency branches*. When the outcomes of such branches depend on loaded values with irregular patterns, branch prediction schemes exploiting correlation in branch histories often fail to predict them accurately. In this paper, we present a novel locality to handle those branches. The new locality is based on the following observation: for some applications, major data structures or key data components tend to remain stable, i.e., the addresses of the key components do not change for a long time. For example, after a linked list is initialized, the address of the ending node remains the same until a new node is appended to it. If a branch is dependent on such stable data, e.g., the branch determining the end of a linked list traversal, the load address instead of the loaded

978-1-4244-2070-4/08/$25.00 ©2008 IEEE

value is sufficient to determine the branch outcome. Therefore, the branch can be resolved once the corresponding load address is known, which is much more promptly than waiting for the loaded value. Since the locality is based on the correlation between load addresses and branch outcomes, we call it address-branch correlation. In a load/branch pair that exhibits address-branch correlation, the load address is referred to as a '*producer address*' and the branch is referred to as a '*consumer branch*'.

In this paper, we present a study on address-branch correlation for memory-intensive applications given the importance of long-latency hard-to-predict branches in those applications. An in-depth analysis of benchmark source codes reveals common program patterns that lead to address-branch correlation. We analyze the performance potential of exploiting address-branch correlation and find that a large portion of branch misprediction penalties can be eliminated by focusing on a few (3 to 4) hard-to-predict branches that exhibit address-branch correlation. For example, 57% of the total branch misprediction penalties of *mcf* can potentially be eliminated by exploiting address-branch correlation in 4 branches. We then propose an auxiliary branch predictor, named Address-Branch Correlation (ABC) predictor, to exploit the locality. In the ABC predictor, stable address-branch correlation information is stored in a prediction table. When a producer address is known, this prediction table is accessed to see whether the address has stable correlation with a consumer branch. If so, the branch outcome is predicted and the prediction is used as either a prioritized one when the branch has not been fetched or an overriding one when the branch has already been fetched using the prediction from the primary branch predictor. Our experimental results show that augmenting a 16kB TAGE branch predictor [13],[14] with a 9kB ABC predictor reduces the execution time by 6.3% (up to 27%) and the energy consumption by 5.2% (up to 24%) of a set of memory-intensive applications, which outperforms a 64kB TAGE branch predictor.

The paper is organized as follows. Section 2 presents a study of address-branch correlation. The ABC predictor is described in Section 3. The experimental methodology and results are in Sections 4 and 5, respectively. Section 6 highlights the limitation and potential optimizations of the proposed approach. Section 7 concludes the paper.

2. Address-Branch Correlation

In this section, we present an in-depth study of address-branch correlation. We focus on long-latency hard-to-predict branches, i.e., those branches that incur high performance penalties and can not be predicted accurately using existing branch predictors including perceptron branch predictors [6],[7],[18] and recently proposed predictors based on the Prediction by Partial Matching (PPM) algorithm [11],[13]. We observe that the outcomes of many long-latency hard-to-predict branches show stable correlation with their producer load addresses. Therefore, the branch outcome can be predicted accurately using the load addresses instead of waiting for the loaded values. We use source-code examples to reveal the inherent reason why such address-branch correlation exists. We then analyze a set of memory-intensive benchmarks to examine the potential benefit. All simulation results in this section are based on the processor configuration described in Section 4 unless otherwise stated.

2.1. Motivation

We first use a microbenchmark to illustrate the correlation between producer load addresses and consumer branch outcomes. The key data structure in the microbenchmark, as shown in Figure 1, is a linked list. The linked list is initialized to contain 10 nodes. Then it is traversed 100000 times. Before each traversal, the linked list is updated by either inserting or deleting a node at a random position except the ending node. The maximum length of the linked list is restricted to 20. Although completely random behavior is rare in real applications, many hard-to-predict branches exhibit similar irregular behavior to the branches in the microbenchmark.

```
Initialization();
for (i = 0; i < 100000; i++) {
   RandomOp();  //Insert or delete a node at a random
position except the end node
   node = head;
   while( node ) {            //Branch 1
      node = node->next;      //Load 1
   }
}
```

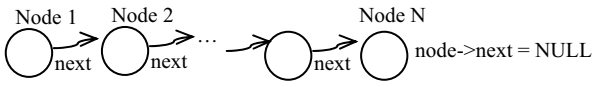

Figure 1. A microbenchmark to illustrate the address-branch correlation.

The branch labeled 'Branch 1' in Figure 1 is the one of interest. It depends on the pointer-chasing load, "node = node−>next" (labeled 'Load 1'), to determine the end of the linked list. If the linked list has N nodes, the while loop will iterate N times and the branch outcomes would be (N–1) 'Takens' followed by one 'Not-Taken'. Because of the random node insertion/ deletion operation, neither gshare [10] nor TAGE branch predictors predict Branch 1 accurately. A 16kB

gshare predictor reports a 9% misprediction rate and a 16kB TAGE predictor has a 6.5% misprediction rate. Furthermore, since Branch 1 depends on Load 1, a misprediction can not be resolved until Load 1 returns the value. If Load 1 has a high L2 cache miss rate, the branch misprediction penalties of Branch 1 will have significant impact on the overall performance. However, as shown in the source code, the traversal always ends at the last node. As long as this node remains stable, i.e., it is not deleted and no new nodes are appended to it, the address of the 'next' field of the node (i.e., the load address of Load 1) is sufficient to determine the outcome of Branch 1. Assuming that the address of the 'next' field of the last node is *X*, Branch 1 would be always taken until the load address of Load 1 becomes *X*. From this example, we can see that strong correlation exists between the address of the producer load and the outcome of the consumer branch and the correlation is a direct result of stable data structures during program execution.

The microbenchmark in Figure 1 highlights the branch-address correlation between a load and a branch in the same loop iteration. We can also extend the scope to extract correlation between load addresses and branch outcomes across different iterations. For example, in the linked list traversal microbenchmark, if the last two nodes (i.e., Node N-1 and Node N) of the linked list are stable, the address of the 'next' field of node N-1 also correlates with the outcome of Branch 1 of the *next* iteration, as shown in Figure 2. In other words, once node N-1 is accessed, it is certain that Branch 1 will be 'Not-Taken' in the next iteration. Correlation across iterations is helpful when both node N-1 and node N accesses (dependent loads) are cache misses. Rather than waiting for the 'next' field of node N to be loaded, the address of node N-1 provides sufficient information to determine the outcome of Branch 1. We use the term '*correlation distance*' to describe address-branch correlation across multiple iterations. Address-branch correlation with a distance of k means that the correlation exists between the load and the branch across k iterations. When the load and the branch are in the same iteration, the correlation distance is zero.

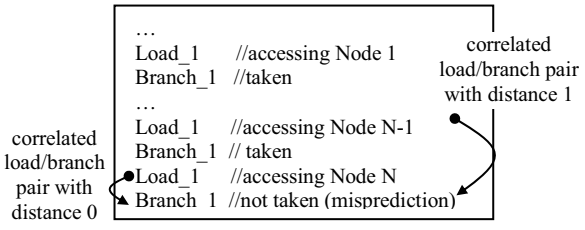

Figure 2. Address-branch correlation with different correlation distances.

2.2. Benchmark Study

After illustrating address-branch correlation using a microbenchmark, we now examine real workloads. As our focus is on long-latency hard-to-predict branches, we select the notorious pointer-intensive benchmark *mcf* from the SPEC CINT 2000 benchmark suite as a representative workload. Using program profiling, we found that the function *refresh_potential* is invoked frequently to refresh a huge tree structure that is larger than typical L2 caches. As a result, the pointer-chasing code in this function has high L2 cache miss rates. Furthermore, the loaded values that determine the conditional branch outcomes are irregular, which makes the branches hard to predict. Overall, frequent cache misses combined with branch mispredictions make this function a critical performance bottleneck of *mcf*. Figure 3 shows the key segments of the function, in both C (Figure 3a) and assembly (Figure 3b).

```
long refresh_potential( network_t *net )
                …
  while( node != root ) {
    while( node ) { //long-latency branch
      if( node->orientation == UP ) //long latency branch
        node->potential = node->basic_arc->cost + node->pred-
>potential;
      else { /* == DOWN */
        node->potential = node->pred->potential - node-
>basic_arc->cost;
        checksum++;
      }
      tmp = node;
      node = node->child;
    }
          …
  }
```

(a) C source code

```
…
00400808 lw $v0[2],28($a0[4])  // Load 1 "node->orientation"
00400810 bne $v0[2],$a3[7],00400848
          // Branch 1 "node->orientation == UP"
00400818 lw $v0[2],32($a0[4])
00400820 lw $v1[3],8($a0[4])
…
00400878 sw $v0[2],44($a0[4])
00400880 addu $v1[3],$zero[0],$a0[4]
00400888 lw $a0[4],12($a0[4]) // Load 2 "node = node->child"
00400890 bne $a0[4],$zero[0],00400808 //Branch 2 "while( node )"
```

(b) Assembly code

Figure 3. A code example extracted from *mcf*.

The code segment in Figure 3b consists of two hard-to-predict branches, labeled as 'Branch 1' and 'Branch 2', respectively. Our simulation results show a 17.4% misprediction rate for Branch 2 using a 16 kB TAGE branch predictor. The average misprediction penalty of Branch 2, measured as the latency between fetching the branch instruction and resolving the misprediction, is as high as 636 cycles due to dependent cache misses resulting from pointer chasing.

Compared to Branch 2, Branch 1 enjoys much lower misprediction rate, 3.1% with a 16kB TAGE branch predictor. However, it still suffers from high misprediction cost, 542 cycles on average. Overall, Branch 1 and Branch 2 contribute 9.5% and 62.1% of the overall branch misprediction penalties of *mcf*.

Either of the two high-cost branches is dependent on a load instruction, labeled as 'Load 1' and 'Load 2' in Figure 3b. A trace containing the load addresses and branch outcomes reveals that both of the load/branch pairs exhibit strong address-branch correlation. The correlation between Load 2 and Branch 2 is similar to the linked list traversal microbenchmark presented in Section 2.1. The load/branch pair (Load 1 and Branch 1) shows a different type of code exhibiting address-branch correlation. Rather than searching for a NULL field, the branch outcome is dependent on comparing the loaded value with a constant non-zero value. By inspecting the source code shown in Figure 3a, we can see why the load/branch pair exhibits strong address-branch correlation. Although the code continuously refreshes the 'potential' field of many nodes, the address of each node and the 'orientation' field of the nodes, which affects the branch outcomes, remain stable. Therefore, once the load address is known, branch outcomes can be determined since those loaded values are not changing over time.

Besides pointer-intensive code, we also found stable address-branch correlation in workloads using other types of data structures. One such example is large sparse matrices. As long as the matrix stays stable (i.e., non-zero entries are not reset to zero), the load address provides enough information to determine the outcome of a dependent branch.

2.3. Performance Potential

If a program exhibits strong address-branch correlation, the load address can be used to either accurately predict a branch or promptly override a misprediction when the branch has already been fetched. In this section, we examine the performance potential of exploiting address-branch correlation.

In our experiments, we select memory-intensive benchmarks from the SPEC2000 benchmark suite using the following criterion: the benchmarks that gain at least 40% performance improvement with an ideal L2 cache are considered memory intensive. We used the reference inputs and simulated 300M instructions for each benchmark. The simulation points are determined using the Simpoint toolset [16] except *parser*. In *parser*, key data structures are refreshed for each new input sentence. The simulation point of parser selected by Simpoint includes the processing of many different sentences, in which stable address-branch correlation only exists for short periods. Therefore, for *parser* we use a simulation point (skip the first 700M instructions) in which most of the time is spent on processing a single complex sentence.

We focus on the branches that have a large number of mispredictions if predicted by a 16kB TAGE predictor. For each of them, we perform the following analysis to examine whether it has correlation with its producer load addresses and how much misprediction penalty can be reduced if such correlation exists. As the first step, we identify the producer loads of each hard-to-predict branch. For each dependent load/branch pair, we then generate a trace of its dynamic instances. For each producer load, the trace contains its load address and the cycle when the address is generated (Addr_cycle). For each consumer branch, the trace includes the cycle when the branch is fetched (Fetch_cycle), the cycle when the branch is resolved (Resolve_cycle), the prediction from the TAGE predictor, and the actual branch outcome.

Based on the trace, we keep track of the correlation between producer addresses and consumer branch outcomes of a certain correlation distance *k*. If a producer address correlates to a consumer branch outcome and the branch is mispredicted, we assume that this misprediction can be eliminated by exploiting address-branch correlation. The misprediction penalty reduction is calculated as *(Resolve_cycle of the consumer branch – Addr_cycle of the producer load of correlation distance k)* if the branch is fetched before the correlated address is available. In this case, the prediction based on the correlated address serves as an early misprediction resolution. If the branch has not been fetched when the correlated address is available, the misprediction penalty reduction is *(Resolve_cycle of the consumer branch – Fetch_cycle of the consumer branch)* assuming that the prediction is used as a prioritized one to direct the instruction fetch unit. The sum of all misprediction penalty reductions is then the performance potential of exploiting address-branch correlation for this load/branch pair of a correlation distance *k*. If a branch has multiple correlated producer loads, we select the one with the highest reduction.

To examine the relationship between correlation distances and potential reductions, we present 4 load/branch pairs with large potential reductions from 4 different benchmarks as shown in Figure 4. The reductions are normalized to the total misprediction penalties of the branch. The figure shows that a large portion of the misprediction penalties could be eliminated if a branch has strong address-branch correlation. The maximum reduction is typically

achieved with a correlation distance larger than 0. Larger correlation distances offer higher reduction since mispredictions can be resolved more promptly. On the other hand, longer correlation distances requires more stable data components, e.g., a correlation distance k requires the last (k+1) nodes in a linked list to be stable rather than just the last node. If data structures fail to provide such high degree of stability, address-branch correlation will drop dramatically with the increased correlation distance, as observed from the load/branch pair from *vpr*.

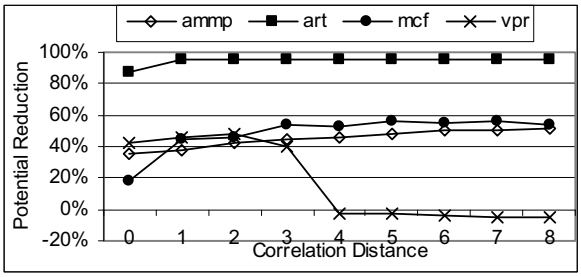

Figure 4. The relationship between potential reduction and correlation distance.

After determining the optimal correlation distance for each load/branch pair, we select the top N (N =1 to 8) pairs from each benchmark. The normalized reductions from these N pairs are reported in Figure 5. The results show that among all the benchmarks, *ammp*, *art*, *mcf* and *parser* show the strongest address-branch correlation. By exploiting address-branch correlation in the top 8 branches, 45% to 57% of the overall branch misprediction penalties could be eliminated. Among the remaining benchmarks, *equake*, *twolf*, and *vpr* show reductions ranging from 11% to 17% while the benchmarks *gcc* and *swim* report very limited address-branch correlation. Another important observation from Figure 5 is that most of the performance benefits can be achieved with only few (3 or 4) load/branch pairs. Since correlated branches and addresses need to be stored in order to exploit address-branch correlation, limiting the number of load/branch pairs will reduce the associated storage requirement.

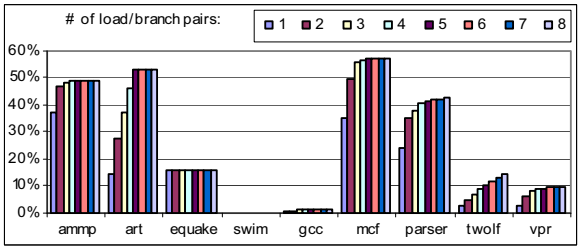

Figure 5. Potential reductions in branch misprediction penalties by exploiting address-branch correlation from the top N (N = 1 to 8) load/branch pairs.

3. An Address-Branch Correlation Based Predictor (ABC predictor)

In order to explore address-branch correlation, we need to identify load/branch pairs showing strong address-branch correlation. Such identification can be done either statically with program profiling or dynamically with hardware support. In this paper, we propose a two-step dynamic approach to capture load/branch pairs with strong address-branch correlation. In step 1, we select the hard-to-predict branches and their producer loads. In step 2, we determine whether the selected load/branch pairs have strong address-branch correlation and what correlation distances are optimal. The detailed hardware implementation of this two-step identification is presented in Section 3.1.

After the load/branch pairs are identified, their PCs and correlation distances stay in a small content-addressable memory (CAM), which is used to check whether a dynamic load or branch instruction needs to access an Address-Branch Correlation based predictor (ABC predictor). Since the proposed ABC predictor is only used to handle those long-latency hard-to-predict branches, we use it as an auxiliary predictor to a primary branch predictor such as a gshare or TAGE predictor.

In our proposed ABC predictor, the prediction process starts with a producer load address, which is computed in the Address Generation (AGEN) stage of a producer load instruction. A prediction table, which keeps track of address-branch correlation information, is then accessed with the producer address. If this address correlates to a consumer branch outcome, the prediction table returns a prediction. Based on whether the consumer branch has been fetched, the prediction can be used as either an overriding prediction to resolve a misprediction or a prioritized one to direct the instruction fetch unit. The prediction table is updated when a consumer branch commits. The detailed design of the prediction table is presented in Section 3.2.

Unlike traditional branch predictors, the proposed ABC predictor involves two correlated instructions, a producer load and a consumer branch. As a result, it is necessary to link the dynamic instances of the load/branch pair in order to determine the producer/consumer relationship. In the proposed design, a FIFO structure, named Address-Branch Correlation Queue (ABCQ), is used for this purpose and the associated operations to link load/branch pairs are discussed in Section 3.3.

3.1. Capturing Load/Branch Pairs with Strong Address-branch Correlation

To capture load/branch pairs with strong address-branch correlation, a two-step approach is used and the associated hardware implementation is shown in Figure 6.

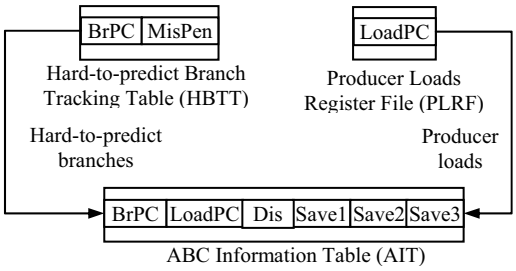

Figure 6. Hardware structures to capture load/branch pairs with strong address-branch correlation.

The first step is to capture hard-to-predict branches and to find their producer loads. To do so, a Hard-to-predict Branch Tracking Table (HBTT) is used to keep track of branches with high misprediction penalties. The HBTT is organized as a 16-entry 4-way cache structure. As shown in Figure 6, each entry has two fields: the branch PC (BrPC) and its accumulative misprediction penalties (MisPen). This table is accessed when a mispredicted branch is retired. If the branch hits in the table, its misprediction penalty is added into the MisPen field. If the branch misses in the table, a new entry is allocated. The misprediction penalty is measured in the following way. Besides the renaming table, each shadow map saves the timestamp when a branch is dispatched. When the branch is resolved, the difference between the current timestamp and the one stored in the shadow map is the latency to resolve this branch. This latency is carried with the branch and used at the retire stage to update the HBTT. After training the HBTT for 2M instructions, a branch is determined hard-to-predict if it has an accumulative misprediction penalty larger than 10000 cycles. If there is at least one such branch, we stop training the HBTT and select out 5 hard-to-predict branches. Otherwise, we clear the HBTT and train it for another 2M instructions. We performed experiments with varying the training period and found that training 2M instructions provides a good balance between the quickness in capturing hard-to-predict branches and the accuracy to capture the most important ones. The PCs of those selected hard-to-predict branches are copied into the ABC Information Table (AIT) and we start to identify the producer load for each branch in the AIT using a structure named Producer Loads Register File (PLRF) in Figure 6. The PLRF tracks the dynamic data dependency among instructions. It is indexed by the logical register number and the content of each entry is the PC of the load instruction that affects the corresponding register, either directly or indirectly. The PLRF works as follows. When an instruction is retired, it updates the PLRF indexed by its destination register. If it is a load, it updates the PLRF entry with its PC. If the instruction is not a load but has a destination register, the PCs stored in the PLRF entries corresponding to its source registers are read out. Then the PC, which is closer to the current instruction's PC, is selected to update the PLRF. For example, if an add instruction with the PC 0x4000ABC8 retires, the PLRF entries corresponding to its two source registers contain 0x4000ABC0 and 0x4000ABB0. 0x4000ABC0 will be used to update the PLRF as it is closer to 0x4000ABC8 than 0x4000ABB0. If a register-writing instruction has no source registers (or using r0, the constant zero), we reset the entry in the PLRF entry corresponding to its destination register. This way, when a branch that hits in the AIT is retired, it can access the PLRF with its source registers to determine its producer load.

The second step is to determine whether the selected load/branch pairs have strong address-branch correlation and which correlation distances are optimal. To do so, for each selected load/branch pair in AIT, we keep track of the benefit by exploiting address-branch correlation with three pre-selected correlation distances (1, 3 and 5). We select those three correlation distances since most load/branch pair's optimal distances are from those three or very close to one of them. Each distance is utilized by the ABC predictor (see Sections 3.2 and 3.3) for at least 512 hits in the predictor and at least 2M instructions. In this period, we don't actually use the ABC prediction to substitute or override the primary predictor's prediction. We only use the prediction to estimate how much misprediction penalty could be saved. At the retire stage of a selected branch, if the ABC prediction is correct and the primary predictor is wrong, the Save field of this branch is incremented by the misprediction penalty. On the other hand, if the ABC prediction is wrong and the primary predictor is correct, we decrement the Save field. After the benefits of all three correlation distances are estimated, we compare the Save fields and decide which correlation distance is best for a load/branch pair. The PCs and correlation distances of those load/branch pairs remain in the AIT, which are used to check whether a dynamic load or branch instruction needs to access the ABC predictor.

Depending on the density of hard-to-predict branches, the overall training period ranges from 8M instructions (*mcf*, *parser*, *twolf*) to 148M instructions

(*equake*). The length of the overall training period and the number of load/branch pairs captured for each benchmark are reported in Table 1. To evaluate the effectiveness of our dynamic learning scheme, we performed profiling to select out the correlated load/branch pairs and their optimal correlation distances. Our experimental results show that the performance benefit of profiling over the proposed dynamic learning scheme is only 0.7%.

Table 1. Length of the training period and the number of selected load/branch pairs.

	ammp	art	equake	mcf	parser	twolf	vpr
Training (M Insts.)	13	18	148	8	8	8	9
# of selected pairs	1	3	1	3	4	2	3

3.2. Tracking Address-Branch Correlation with a Prediction Table

We use a prediction table to keep track of address-branch correlation information. The prediction table is organized as a 2-way set-associative structure. Each entry of the prediction table contains three fields, a partial 'Tag' containing several bits of a producer address; a 1-bit 'Correlated' flag indicating whether the address has correlation to the branch outcome, and a 1-bit 'Pred' field keeping the outcome of a consumer branch, as shown in Figure 7.

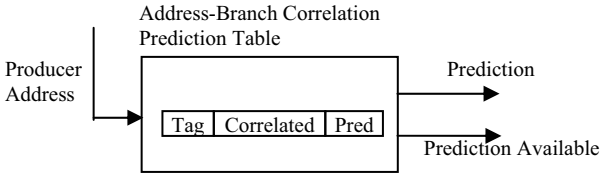

Figure 7. An Address-Branch Correlation Prediction Table.

The prediction table is accessed when a producer address is available. If the producer address hits in the table and the 'Correlated' flag is true, a valid prediction is generated using the 'Pred' field.

When a consumer branch is retired, the prediction table is updated with the producer address (obtained from the ABCQ described in Section 3.3), the actual branch outcome, and whether the branch is mispredicted by the primary predictor. If the producer address misses in the prediction table and the primary predictor has a misprediction, a new entry is allocated in the prediction table, the 'Pred' field is set to the branch's actual outcome, and the 'Correlated' flag is set to true. If the address hits in the prediction table, the 'Correlated' flag is set to false if the actual branch outcome doesn't match with the 'Pred' field, indicating that there is no stable correlation with this address.

When a new entry is required at a set where no empty entry is available, an existing entry is replaced only if its 'Correlated' flag is false. The reason is that we do not want to replace useful correlation information. To overcome the problem of out-dated correlation, which means that some addresses in the prediction table will not be accessed anymore and will not be replaced, we resort to resetting the table periodically at an interval of 100M instructions. Since long-latency mispredictions incur much higher branch penalties than short-latency ones, among the first 2M executed instructions after each reset, only long-latency mispredictions are allocated a new entry in the prediction table. Short-latency mispredictions compete for the rest of the entries afterwards. We treat a misprediction with more than 100-cycle resolution latency as a long-latency misprediction.

3.3. Linking Producer Loads and Consumer Branches

We use Address-Branch Correlation Queues (ABCQs) to link dynamic instances of producer loads and their consumer branches. Each selected load/branch pair has its own ABCQ. Each ABCQ maintains information of dynamic instances of a particular branch that have been fetched but not yet been committed. Each ABCQ entry contains four fields: Prediction Available (PredAvail), Prediction (Pred), Address Available (AddrAvail), and Address (Addr). The first two fields indicate whether a valid ABC prediction is available and what the prediction is whereas the next two fields show whether a valid producer address is available and what the address is. The Pred field is also used to store the primary predictor's prediction if an ABC prediction is not available when a consumer branch is fetched. The queue is managed by three pointers: 'Head', which points to the dynamic instance of the consumer branch that will retire next; 'Dispatch', which points to the instance that will be dispatched next; and 'Fetch', which points to the instance that will be fetched next. Those pointers are updated when a dynamic instance of the corresponding branch is retired, dispatched and fetched, respectively. The reorder buffer (ROB) is also augmented with an 'ABCQ_ptr' field, which provides a link to the corresponding ABCQ entry. The queue structure and its relationship with ROB are shown in Figure 8.

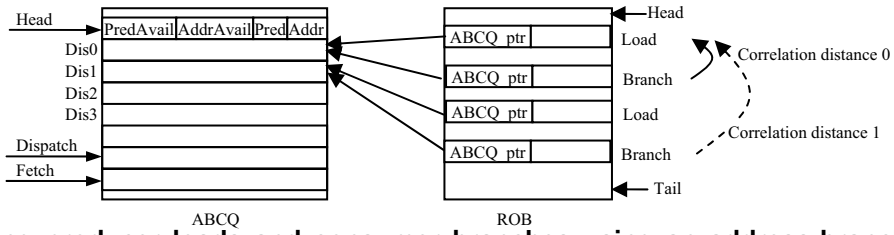

Figure 8. Linking producer loads and consumer branches using an address-branch correlation queue (ABCQ).

When a producer load of interest is dispatched, the current Dispatch pointer of the corresponding ABCQ is stored in the ABCQ_ptr field of its ROB entry. When its consumer branch is dispatched, the same pointer is stored in its ABCQ_ptr field, as shown in Figure 8. Since the ABCQ maintains the branch's dynamic instances in the program order, it provides an efficient way to link producer loads and consumer branches across multiple iterations. For example, a load can reach its correlated branch with correlation distance 1 using (ABCQ_ptr + 1) % ABCQ_size, as illustrated in Figure 8.

When the address of a producer load is known, it is used to access the prediction table to retrieve an ABC prediction. The prediction as well as the address will be used to update the ABCQ entry through the ABCQ_ptr. In other words, the entry ABCQ[(ABCQ_ptr + k) % ABCQ_size] is updated if address-branch correlation of distance k is to be exploited for this load instruction. Since the Pred field may also be used to store the primary predictor's prediction, we need to check the current 'Fetch' pointer of the ABCQ before updating the field with the ABC prediction. The 'Fetch' pointer of the ABCQ shows whether the dynamic instance of the consumer branch has been fetched or not. If it has been fetched, the fetch pointer will exceed (ABCQ_ptr + k) and the Pred field maintains the existing prediction generated by the primary predictor. In this case, the Pred field is compared with the ABC prediction to decide whether to override the existing prediction and invoke a misprediction recovery. Then, the ABC prediction will be stored in the Pred field. If the dynamic instance has not been fetched, the PredAvail flag will be set to true and the ABC prediction will be used as a prioritized prediction when it is fetched, i.e., when the 'Fetch' pointer equals to (ABCQ_ptr + k).

When a consumer branch is ready to commit, its actual outcome as well as the information maintained in its assigned ABCQ entry, pointed through its ABCQ_ptr, will be used to update the prediction table. After the branch is committed, its ABCQ entry is de-allocated.

Since an ABCQ allocates an entry for each fetched instance of a particular branch, some of those dynamic instances may be on wrong paths. In order to recover the ABCQs in case of branch mispredictions, their Dispatch and Fetch pointers are checkpointed along with the rename map table when a branch is dispatched. When a branch misprediction is detected, the checkpointed pointers are used to recover the ABCQ's state.

3.4. Hardware Cost

The storage requirement of our design includes two parts: the hardware to capture load/branch pairs and the ABC predictor. The HBTT has 16 entries and each entry includes a 32-bit PC and a 24-bit saturating counter to store the misprediction penalty. The AIT has 5 entries and each entry includes two 32-bit PCs, a 3-bit counter to store correlation distance and three 21-bit saturating counters. The PLRF has 64 entries and each entry stores a 32-bit PC. The overall storage requirement of those three tables is 3594 bits (449 bytes).

The storage requirement of the proposed ABC predictor depends on the sizes of the prediction table and ABCQs. In our design, we use a 7-bit partial tag. Therefore, each entry in the prediction table has 9 bits, including the correlation and prediction flags. A larger prediction table can capture more correlated addresses, but results in larger hardware cost and higher access latency. We experimentally found that a 9kB 2-way associative table achieves good balance between performance and hardware cost (See Section 5.1).

Each ABCQ entry has three 1-bit fields and an address field. Since the address field is used to access the prediction table, only the index and the partial tag need to be stored. For a 9kB 2-way prediction table, the index needs 12 bits and the partial tag has 7 bits. Therefore, each entry of the ABCQs has 22 bits. We use 64-entry ABCQs since our results show that the performance with 64-entry ABCQs is very close to the performance of queues with infinite number of entries. The reason is that the selected branches are hard to predict. If a large number of these branches have been

fetched, it is very likely that the front end is already on a wrong path and there is no performance loss to stall the front end. Considering the requirement of training 5 load/branch pairs as described in Section 3.1, we use 5 ABCQs and the total cost of the queues is 7040 bits (880 bytes).

4. Simulation Methodology

Our simulator infrastructure is built upon the SimpleScalar toolset [3] but our execution-driven timing simulator is completely rebuilt to model MIPS R10000-style out-of-order superscalar processors. The processor configuration is shown in Table 2. We model a large instruction window processor to highlight the performance impact of branch mispredictions since the memory wall problem can be resolved effectively with large instruction windows. The primary branch predictor is a 16kB TAGE predictor which uses 130-bit global branch histories with ideal 1-block ahead configuration. The performance of the proposed ABC predictor is evaluated by using SPEC 2000 benchmarks with the reference inputs and the same selection criterion and simulation points as described in Section 2.3. The dynamic load/branch pairs selecting period is also included in the total simulation range of 300M instructions. We exclude *swim* and *gcc* because their estimated branch misprediction penalty reductions are too small (less than 2%) as shown in Figure 5.

Table 2. Configuration of the processor.

Instruction Cache	32 kB, 2-way; Line size=16 Inst.; Miss penalty=*10* cycles.
Data Cache	32 kB, 2-way; Line size = 64 bytes; Miss penalty =*10* cycles.
Unified L2 Cache	1024 kB, 8-way; Line size=128 bytes; Miss penalty=300 cycles.
Primary Branch Predictor	16kB TAGE: 8 prediction tables. Min branch mispred. penalty = 20 cycles.
Superscalar Core	Reorder buffer: 1024 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units; Data cache ports: 4. Issue queue: 512 entries. LSQ: 512 entries. Rename map table checkpoints: 256.
Execution Latencies	Addr. Gen.: 1 cycle; Mem. access: 2 cycles (hit in data cache); Int. ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies.
Memory Disambiguation	Perfect memory disambiguation.
Hardware Prefetcher	Stride-based stream buffer: 8 four-entry stream buffers with a PC-based two-way 512-entry stride prediction table.

5. Experimental Results

5.1 Performance

In this experiment, we examine the performance of our proposed ABC predictor. We augment the baseline processor with an ABC predictor and vary the size of its prediction table from 4.5kB to 72kB. The normalized execution time with respect to the baseline processor is reported in Figure 9. For reference, we also include the results of using ideal predictions for those selected branches.

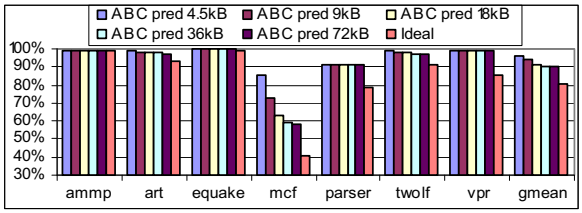

Figure 9. Normalized execution time of the baseline processor augmented with ABC predictors.

Two observations can be made from Figure 9. First, among the memory-intensive benchmarks that we examined, the ABC predictors always result in positive performance improvements and higher performance improvements are achieved in integer benchmarks than floating-point benchmarks, which is expected as even ideal prediction has limited performance impact in floating-point benchmarks. Among the integer benchmarks, *mcf* and *parser* show strong address-branch correlation which confirms the performance potential study shown in Figure 5. Second, although larger prediction tables can capture more address-branch correlation information and result in better performance, a small 4.5kB prediction table is sufficient to keep track of the majority of useful producer addresses. An exception is the benchmark *mcf*, for which increasing the table size has significant performance impact, the execution time is reduced by 15% with a 4.5kB table and by 41% with a 72kB table. The reason is that its large working set and pointer-chasing code result in a large number of producer addresses that correlate to consumer branches. On average, the ABC predictor reduces the execution time by 4.1% with a 4.5kB prediction table and 9.5% with a 72kB table. Among the different prediction table sizes, a 9kB prediction table reduces the execution time by 6.3% (up to 27%), which provides a good tradeoff between performance improvement and hardware cost.

5.2 Prediction Accuracy and the Reduction in Branch Misprediction Penalties

In this experiment, we examine the prediction accuracy achieved by the proposed ABC predictors and what fraction of the branch misprediction penalties can be eliminated. For those selected branches a 9kB ABC predictor achieves 96.8% prediction accuracy, which is much higher than a 16kB TAGE predictor (87.8%) and the idealistic GTL predictor [15] with nearly unlimited storage budget (91.8%). However, since the ABC predictor only provides a prediction when a producer load address is highly correlated with the branch outcome, it does not cover all dynamic instances of those selected branches. For a fair comparison, we report in Table 3 the number of mispredictions of those selected branches using a 16kB TAGE predictor with and without a 9kB ABC predictor. The ABC predictor reduces the mispredictions of those selected hard-to-predicted branches by 37.7% on average.

Next, we examine the reduction in overall misprediction penalties achieved by the ABC predictor. We use the interval between the time when a mispredicted branch is fetched and the time when the misprediction is resolved as the misprediction penalty. The reductions are normalized to the misprediction penalty of the baseline processor with a 16kB TAGE predictor and are shown in Figure 10. We can see that although the ABC predictor only provides predictions for one to four (static) selected branches, the proposed ABC predictors successfully reduce overall misprediction penalties and a 9kB ABC predictor can achieve up to 29% and an average of 11% reduction.

5.3 Impact of Primary Branch Predictors

In this experiment, we vary the primary predictor to evaluate its impact on the proposed ABC predictor. We simulate gshare predictors, which are less accurate but relatively easy to implement, and large TAGE predictors, which provide state-of-art prediction accuracy but have higher implementation complexity. The execution time of different branch predictors, either used alone or combined with a 4.5kB or a 9kB ABC predictor, is shown in Figure 11. Here, the execution time is normalized to the baseline processor with a 16kB gshare predictor.

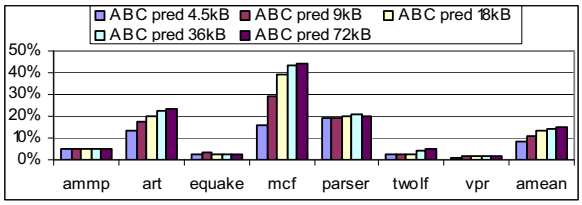

Figure 10. Reductions in branch misprediction penalties achieved by ABC predictors.

From the figure, we can see that the ABC predictor achieves significant performance improvement even if it is used together with a highly accurate primary predictor. When augmenting a 128kB TAGE predictor with a 9kB ABC predictor, the execution time is reduced by 4.2% (up to 15%). For a less accurate predictor, such as a 16kB gshare predictor, the ABC predictor is also effective and reduces the execution time by up to 22% and 5.6% on average. Another observation from the figure is that an 8kB TAGE predictor with a 4.5kB ABC predictor outperforms a 32kB TAGE predictor and a 16kB TAGE predictor with a 9kB ABC predictor outperforms a 64kB TAGE predictor for those benchmarks. The reason is that the ABC predictor achieves high prediction accuracy for a few long-latency branches while the larger TAGE predictor aims to improve prediction accuracy universally for both long-latency and short-latency branches. Focusing on long-latency branches is more effective in reducing the overall execution time.

Table 3. The number of mispredictions for the selected hard-to-predict branches.

	ammp	art	equake	mcf	parser	twolf	vpr	amean
16kB TAGE	58005	257476	197533	2842260	478172	577130	522262	704691
16kB TAGE + 9kB ABC	24388	117594	191520	1159779	179864	458304	489888	374476
Misprediction Reduction	58.0%	54.3%	3.0%	59.2%	62.4%	20.6%	6.2%	37.7%

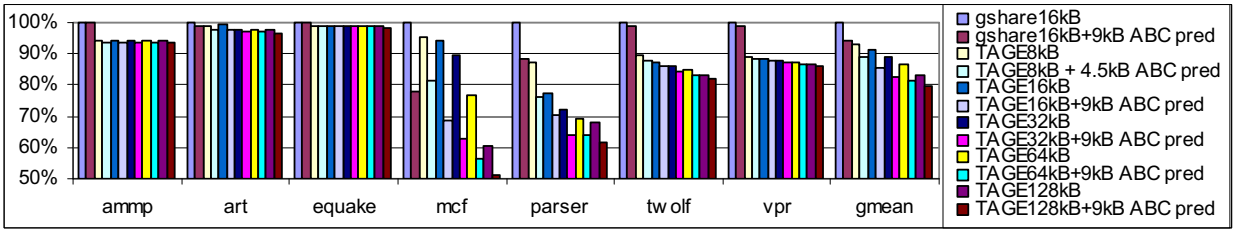

Figure 11. Normalized execution time of different primary predictors with and without ABC predictors.

(a) ROB Size *(b) L2 Size* *(c) Memory Latency (cycles)*

Figure 12. Performance improvements for processors with different ROB sizes, L2 sizes and memory latencies.

5.4 Sensitivity Study

In this experiment, we study the sensitivity of the proposed ABC predictor on various architecture parameters. We use the baseline configuration shown in Table 2 and vary the ROB size, the L2 cache size and the memory latency. The performance improvements of a 9kB ABC predictor calculated by the reduction of the geometric mean of the normalized execution time are shown in Figure 12. As shown in Figure 12a, the ABC predictor provides larger benefit to processors with larger instruction windows. The reason is that a large instruction-window processor can effectively exploit memory-level parallelism to overcome the memory wall problem. Therefore, the pressure is essentially shifted to branch predictors to fetch a large number of instructions from correct paths. The percentage of the total misprediction penalty to the execution time increases from 24% to 30% when we increase the ROB size from 128 to 1024. For the impact of the cache and memory system as shown in Figure 12b and Figure 12c, we can observe that the ABC predictor provides similar performance benefits (5.5% to 6.5%) with different L2 sizes and memory latencies. Generally, with a larger L2 size or smaller memory latency, the performance improvement is higher since the performance bottleneck is shifted from the latency of memory operations to the branch prediction accuracy. An exception is the *mcf* with a processor having a 4MB L2 cache. Compared to a 2MB L2 cache, the L2 cache miss rate of the 4MB L2 cache drops dramatically for *mcf* (from 46% to 26%). Such drop reduces the relative importance of the load dependent branches predicted by the ABC predictor and the performance improvement of *mcf* is reduced from 26% to 19%.

5.5 Reduction in Energy Consumption

Since long-latency branch mispredictions lead to a large number of wrong-path instructions being executed, a reduction in mispredictions reduces the energy being wasted by wrong-path instructions. To analyze the energy consumption effect of the ABC predictor, we port WATTCH [2] and HotLeakage [19] into our simulator to account for both dynamic and static energy consumption.

In our experiments, we use the 70nm technology with a clock frequency of 5.6GHz and assume the linear clock gating [2]. The energy consumed by the ABC predictor is taken into account by modeling the HBTT, the AIT, the PLRF, the prediction table and ABCQs. Since the ABC predictor is only accessed by selected load/branch pairs, its energy consumption is very small and accounts for less than 0.6% of the total energy consumption of the processor. The overall energy consumption of a processor with a 9kB ABC predictor normalized to the baseline processor is shown in Figure 13. From the figure we can see that the ABC predictor reduces energy consumption by up to 24% in *mcf* and 5.2% on average. For *equake*, the energy consumption is increased by 0.4% because the extra energy consumed by the ABC predictor is larger than its contribution to energy reduction. The energy reductions mainly come from the reduced execution time and the reduced number of instructions fetched and executed by the processor. Our results show that the ABC predictor reduces the number of the fetched instructions by 6.5% and the executed instruction by 2.8%.

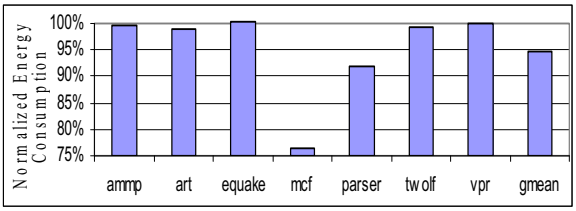

Figure 13. Reduction in energy consumption achieved by a 9kB ABC predictor.

6. Limitations and Potential Optimizations

The proposed approach is a specialized approach aiming to exploit the address-branch locality. Our results show that it is effective for benchmarks with heavy pointer chasing since branches with strong address-branch correlation in those benchmarks normally contribute to a large portion of the total

mispredictions. However, if the relative importance of those branches is low, there is limited performance benefit. For those benchmarks, when exploring address-branch correlation is not effective, we can turn off the ABC predictor or reuse the prediction table to predict other branches. Since each entry of the prediction table has a partial tag and two 1-bit fields, we may reuse the table as a tagged branch prediction table working with the primary branch predictor to improve prediction accuracy of all conditional branches.

7. Conclusions

In this paper we present a novel locality named address-branch correlation (ABC) that can be exploited to handle long-latency hard-to-predict branches. A detailed study of address-branch correlation reveals why stable address-branch correlation exists. The reason is that in many memory-intensive workloads, major data structures or key data components that affect branch outcomes tend to remain stable. If a hard-to-predict branch depends on such stable data, the address of the data rather than the data value is sufficient to determine the branch outcome. Since load addresses are obtained much earlier than loaded values, a misprediction can be detected more promptly especially if the loads result in long-latency cache misses.

We then propose a design to exploit address-branch correlation to reduce misprediction penalties of those hard-to-predict long-latency branches. The proposed predictor dynamically captures correlations between producer load addresses and consumer branch outcomes. Address-branch correlation based predictions are generated when a producer address is known and the prediction is used as either an overriding prediction if the branch has been fetched or a prioritized one if the branch has not been fetched. Our experimental results show that an ABC predictor achieves very high prediction accuracy (96.8%) for those hard-to-predict branches. With a 9kB prediction table, the proposed ABC predictor reduces execution time by 6.3% on average (up to 27%) and also reduces energy consumption by 5.2% on average (up to 24%) for a set of SPEC 2000 benchmarks.

Acknowledgement

We thank the anonymous reviewers for their valuable comments on improving our paper.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan, Checkpoint processing and recovery: towards scalable large instruction window processors, *MICRO-36*, 2003.

[2] D. Brooks, V. Tiwari, and M. Martonosi, Wattch: a framework for architectural-level power analysis and optimization, *ISCA-27*, 2000.

[3] D. Burger and T. Austin, The SimpleScalar tool set, v2.0, *Computer Architecture News, vol. 25*, June 1997.

[4] R. S. Chappell, F. Tseng, A. Yoaz, Y. N. Patt, Difficult-path branch prediction using subordinate microthreads, *ISCA-29*, 2002.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero, Out-of-order commit processors, *HPCA-10*, 2004.

[6] D. Jimenez and C. Lin, Dynamic branch prediction with perceptrons, *HPCA-7*, 2001.

[7] D. Jimenez and C. Lin, Neural methods for dynamic branch prediction, *ACM Trans. on Computer Systems*, 2002.

[8] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, A large, fast instruction window for tolerating cache misses, *ISCA-29*, 2002.

[9] Y. Ma, H. Gao, M. Dimitrov, and H. Zhou, Optimizing dual-core execution for power efficiency and transient-fault recovery, IEEE Trans. on Parallel and Distributed Systems, 2007.

[10] S. MacFarling, Combining branch predictors, *Technical Report, DEC*, 1993.

[11] P. Michaud, A PPM-like, tag-based branch predictor, *Journal of Instruction Level Parallelism*, 2005.

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

[13] A. Seznec, P. Michaud. A case for (partially) tagged Geometric History Length Branch Prediction. *Journal of Instruction Level Parallelism, vol. 8,* February 2006.

[14] A. Seznec. A 256 Kbits L-TAGE branch predictor. In The 2nd JILP Championship Branch Prediction Competition (CBP-2), 2006.

[15] A. Seznec. Looking for limits in branch prediction with the GTL predictor. In The 2nd JILP Championship Branch Prediction Competition (CBP-2), 2006.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, Automatically characterizing large scale program behavior, *ASPLOS-X*, 2002.

[17] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, Continual flow pipelines, *ASPLOS-11*, 2004.

[18] L. Vintan, Towards a high performance neural branch predictor, IJCNN-99, 1999.

[19] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, Hotleakage: a temperature-aware model of sub-threshold and gate leakage for architects, *Tech. Reports CS-2003-05, U. Va. Dept. of CS*, 2003.

[20] H. Zhou, Dual-core execution: building a highly scalable single-thread instruction window, *PACT'05*, 2005.

[21] http://camino.rutgers.edu/cbp2/, the 2nd JILP Championship Branch Prediction Competition (CBP-2) official site, 2006.