Power Efficient Branch Prediction through Early Identification of Branch Addresses*

Chengmo Yang and Alex Orailoglu
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
{c5yang, alex}@cs.ucsd.edu

ABSTRACT

Ever increasing performance requirements have elevated deeply pipelined architectures to a standard even in the embedded processor domain, requiring the incorporation of dynamic branch prediction subsystems to hide the execution latency of control-altering instructions. In this paper a low power early branch identification technique which enables the design of extremely power-efficient branch predictors and BTBs is proposed. Through static extraction of program information regarding the distance to subsequent branches, this technique enables the calculation of the next branch address as soon as the direction of the current branch has been predicted. Early identification of branch addresses enables a complete elimination of the power hungry BTB lookups normally occurring at every execution cycle, as well as a just-in-time wake-up mechanism when accessing "hibernating" entries in complex predictors, switched to power-saving mode to reduce leakage power dissipation. A cost-efficient Branch Identification Unit (BIU) to calculate branch addresses is presented and analyzed in terms of power and timing characteristics. The effectiveness of the proposed BTB access policy and predictor wake-up mechanism is also confirmed by the simulation results of the SPECint 2000 and Mediabench benchmarks.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

low-power design, dynamic branch prediction, application-specific processors

*This work is supported in part by NSF Grant 0082325.

CASES'06, October 23–25, 2006, Seoul, Korea.

1. INTRODUCTION

In the last half a dozen years, increasing sensitivity to power consumption has come to constitute one of the defining challenges of processor architecture design. For embedded processors which typically have constraints of battery life and heat dissipation, energy efficiency has even been well established as an important product quality characteristic, as it may severely undermine the usability and acceptance of the product. Consequently, techniques to minimize power consumption of embedded processors are of significant importance in achieving high product quality.

The ever increasing performance requirements have elevated deeply pipelined architectures to a standard even in the embedded processor domain. However, changes in the execution flow caused by branch, jump, and subroutine call instructions introduce significant complications to the efficient utilization of the deep pipeline. Because the instructions to be fetched after a branch instruction are determined by the outcome of the branch, the processor front-end has to wait until the branch is resolved before continuing to fetch from the correct location. This implies a significant performance degradation, as precise information regarding the branch direction and the branch destination are typically available late in the pipelined execution. To alleviate this problem, dynamic *branch predictors* (BPs) and *branch target buffers* (BTBs) have been employed in high-end general-purpose processors to predict branch direction and to cache branch destination, respectively. However, these techniques have not been widely adopted in embedded processors, mainly due to their significant hardware complexity and power consumption.

To effectively exploit the deeply pipelined architectures in current embedded processors, an extremely power-efficient dynamic branch prediction subsystem is required. This constitutes a challenging task, as both the system's performance and the overall energy consumption heavily depend on branch prediction accuracy. While it would seem at first glance that using smaller predictors would result in reduced power dissipation, simply reducing the size of the branch predictor also increases the probability of *aliasing* (two branches improperly affecting each other by mapping to the same entry in the predictor), reducing in turn prediction accuracy and hence system performance. In fact, such a localized reduction may actually increase total energy consumption by making programs run longer [1].

In this paper, an approach for designing a power-efficient branch prediction subsystem while still maintaining high prediction accuracy is presented. Through static extrac-

tion of program information regarding the distance between the first instruction of each basic block and the corresponding subsequent branch, a cost-efficient *Branch Identification Unit* (BIU) is proposed to calculate the next branch address as soon as the direction of the current branch has been predicted. By incorporating this early branch identification technique into a traditional branch prediction subsystem, the power hungry BTB lookups for non-branch instructions normally occurring at every execution cycle are completely eliminated. Moreover, the whole branch predictor can be switched into a power-saving "hibernation" mode to reduce leakage power dissipation. The proposed early branch identification technique additionally enables a just-in-time wake-up mechanism to overcome the potential degradation in prediction accuracy caused by accessing hibernating entries. Consequently, our framework enables significant power savings in both the BTB and the branch predictor, with negligible reduction in prediction accuracy.

The remainder of this paper is organized as follows. Section 2 reviews previous power reduction techniques for branch prediction subsystems. Section 3 discusses the technical motivation in detail. The proposed power and performance efficient branch prediction subsystem as well as the corresponding hardware implementation are presented in sections 4 and 5, respectively. Section 6 provides simulation results, while section 7 offers a brief summary of this paper.

2. PREVIOUS WORK

In the last half a dozen years, increasing research attention has been paid to the power/performance exploration of branch prediction schemes.

Several techniques have been proposed to filter the accesses to the BTB. In [1], a simple hardware module, namely a *prediction probe detector* (PPD), is employed to store some pre-decoded bits to indicate whether a cache line contains conditional branches or not. Unnecessary accesses to the BTB and the predictor are eliminated, yet at the cost of accessing this SRAM module every cycle. An *application customizable branch target buffer* (ACBTB) is proposed in [2], which records for each branch the ACBTB indices corresponding to the two possible subsequent branches to determine in advance the ACBTB entry of the upcoming branch. Another compiler technique is proposed in [3] to filter BTB accesses in VLIW architectures. To inform the processor that a branch is forthcoming, a configurable hint instruction that anticipates the branch address is inserted into the program. However, to avoid degrading performance through increasing the number of long instructions, these hint instructions only substitute existing NOPs in the long instruction slots, thus limiting the effectiveness of this technique.

As for branch predictors, current low power techniques have mainly focused on the reduction of the number of accesses to subpredictors in a combined branch predictor.[1] In [4], a small SRAM module is employed to record the subpredictors used by recent branches, based on which accesses to subpredictors are selectively blocked to achieve power reduction. The same authors extend their work in [5] by furthermore filtering well behaved simple branches from accessing the complex hierarchical branch predictor. Both techniques require dedicated hardware support to record program behavior dynamically. As a comparison, the authors of [6] employ the compiler to characterize branch prediction demands. Each application is partitioned into modules for static profiling, whose results are utilized during dynamic execution to disable subpredictors of the combined branch predictor as well as to resize the BTB.

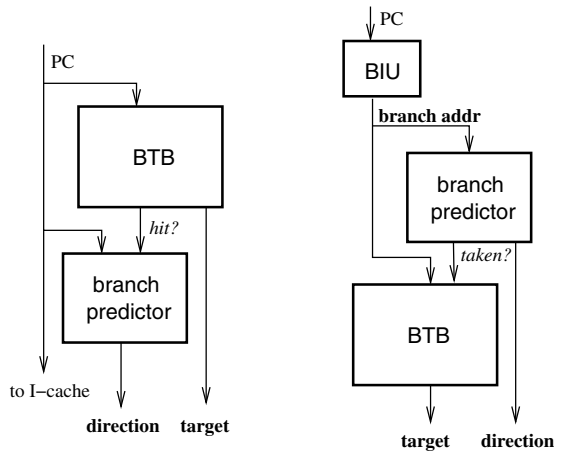
A leakage power reduction technique originally proposed in [7] is adapted to BPs and BTBs in [8]. Entries in BPs and BTBs are turned off if they are not accessed for a certain number of cycles, monitored by a dedicated counter per SRAM row. Obviously, the additional power consumed in these counters limits the amount of energy savings achieved by this technique. More importantly, this technique introduces sizable degradation in prediction accuracy when accessing an entry in hibernation, since the original information regarding previous branch outcomes gets lost.

[1]A combined branch predictor employs a hierachical prediction mechanism; multiple subpredictors with various indexing mechanisms are employed to generate multiple predictions, while at a higher level another predictor is employed to select which subpredictor's outcome is to be used.

3. TECHNICAL MOTIVATION

Designing a power-efficient branch predictor with minimal negative performance impact requires a closer look at the underlying principle of the state-of-the-art branch prediction scheme. Fundamentally, control-altering instructions, such as branches, have long been known to introduce significant performance degradation to pipelined processor architectures. To hide the latency of executing each branch, both the branch direction and the target address should be available to the processor front-end in advance before fetching is resumed from the correct program location. Accordingly, modern pipelined architectures usually predict the branch direction dynamically using BPs, while caching the branch target addresses in BTBs. A typical access flow chart is presented in Figure 1a. Fundamentally, the purpose of the BTB is to provide early branch identification, that is, to determine whether an instruction under fetch is a branch or not. This implies that the BTB has to be always looked up during the first fetch cycle for each instruction, even if for non-branch instructions. As ideally only the branches predicted to be taken need to look up the BTB to obtain their target addresses, this access policy results in a significant amount of power waste in BTB accesses.

Unlike the BTB, branch predictors can be accessed either in parallel with the BTB for each instruction to hide access latency, or in later pipeline stages but only for branch instructions that are already identified through BTB accesses, as shown in Figure 1a. Nonetheless, there exists another source of power inefficiency, as the hardware complexity and hence the associated leakage power dissipation of the predictors is non-trivial. As semiconductor technology advances towards deep submicron, threshold voltages have been lowered to the point where leakage power becomes an important and growing fraction of total power dissipation. More specifically, the elimination of the aforementioned potential aliasing problem tends to enlarge predictor sizes to ensure a sufficiently high prediction accuracy, although for each predictor access only one entry of data is needed. This inefficiency in leakage power dissipation becomes even more significant when hierarchical, multi-level predictors are employed, thus necessitating accesses to multiple subpredictors for generating a prediction for each branch [10].



(a) Traditional branch prediction subsystem

(b) Use of BIU to identify branch addresses

Figure 1: Various BTB access flow charts

In order to eliminate unnecessary BTB accesses, the BTB needs to be freed from the constraint of having to identify branch instructions early. In other words, other units should inform the processor of the subsequent branch address in advance. In this paper, this is achieved through the usage of a cost-efficient ***Branch Identification Unit*** (BIU). More specifically, we propose a technique to statically extract program information regarding *branch distance* for each basic block, that is, the distance between the beginning of that basic block and the corresponding subsequent branch. During dynamic execution, the extracted program information is transferred to the BIU, which calculates the next branch address as soon as the direction of the current branch has been predicted. This early branch identification technique enables the elimination of BTB accesses not only for non-branch instructions, but also for branches predicted to be not-taken. As shown in Figure 1b, if the calculated branch address indicates that the next branch is not forthcoming, an access to the dynamic branch predictor can be completed at first, enabling the BTB to be only looked up if the branch is predicted to be taken. In fact, the ability to eliminate BTB accesses for non-taken branches constitutes the most significant advantage of our framework over previous BTB filtering techniques [1, 2, 3].

As for branch predictors, because only one entry of data is necessitated for each predictor access, if idle entries in BPs can be identified, they can be switched into a power-saving mode to reduce leakage power. Previous techniques have been proposed to dynamically identify inactive SRAM entries for power reduction [7, 9]. However, the lack of a policy for pre-activating entries in hibernation limits the applicability of these techniques to BPs, as accesses to hibernating BP entries result in sizable reduction in prediction accuracy [8]. To overcome the consequent performance degradation, in this paper we propose a prefix analysis technique to cleverly activate "hibernating" rows in a branch predictor that will be accessed relatively shortly. Since the proposed technique can analyze both global branch histories and branch addresses provided by our BIU, it is effective for BPs with various index mechanisms, for example, pure branch PC [11], pure global history [12], or a combination of both [13].

4. PROPOSED FRAMEWORK

4.1 Identifying incoming branches

In order to filter BTB lookups for non-branch instructions as well as to activate hibernating entries in branch predictors, the front-end needs to know in advance the address of the subsequent branch. To exploit maximum power reduction without incurring additional latency in accessing the BP and the BTB, a branch identification mechanism needs to be established in which the next branch address is calculated as early as possible.

Fundamentally, information regarding which branch is to be encountered next during execution is not determined until the current branch direction is resolved. In other words, the earliest point to completely identify the next branch is the instruction just following the current branch. In either the fall-through path or the target path case, the determinative instruction is located at the beginning of a new basic block. Consequently, if the information regarding the distance to the subsequent branch is available at the beginning of each basic block, the next branch address can be identified non-speculatively as early as possible. In this paper, the distance in terms of the number of instructions between the first instruction of a basic block and the corresponding subsequent control-altering instruction is denoted as *branch distance*. More specifically, because a *basic block* (BB) is a linear sequence of instructions with single entry and single exit points, the subsequent branch for BB_i is the last instruction of $BB_j (j > i)$, if none of BB_i, BB_{i+1}, ..., BB_{j-1} but BB_j ends up with a control-altering instruction.

We propose to statically analyze the application to extract control-flow information regarding the branch distance for each BB. This process can be illustrated more clearly by considering an example loop containing three conditional statements presented in Figure 2a. Figures 2b and 2c present the corresponding control structure in the form of a *control flow graph* (CFG) and a possible code layout including 6 control-altering instructions, respectively. It can be observed from Figure 2c that the size of each BB is known immediately after compiling the program, implying that the branch distance for each BB is also determined. For instance, the branch distance of $B2$, which is the distance between its first instruction and *branch2*, is $|B2| - 1$ (with $|B|$ denoting the number of instructions within a basic block B). Similarly, the branch distance of $B6$ is $|B6| + |B7| - 1$, since the corresponding subsequent branch is the *loop branch*. For $B3$ which ends up with an unconditional jump, the branch distance is $|B3| - 1$, because an unconditional jump still needs to look up the BTB to obtain its target address before being executed.

For each application, the extraction of branch distance information is performed during the compile/link time independently for each hot spot, identified through profiling or extracted from the algorithmic specifications. Typically the code size of each hot spot is relatively small as it is well known that 90% of the execution cycles are spent on 10% of the code, resulting in a relatively small number of BBs to be targeted. These various hot spots are furthermore extremely independent, making it possible to apply local

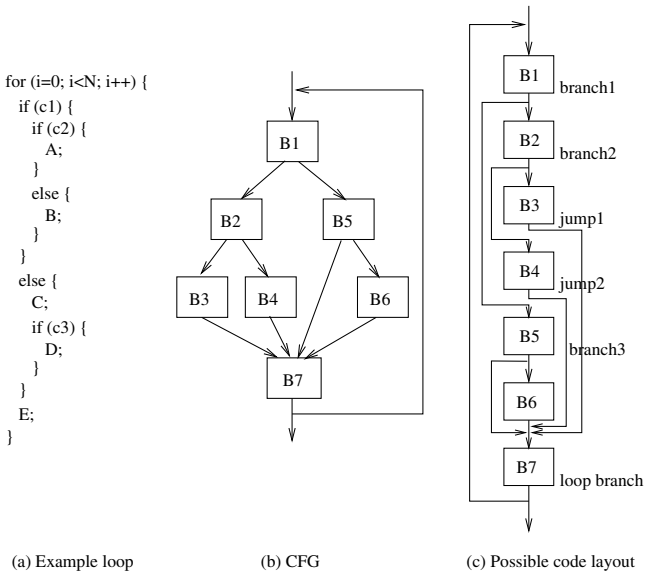


Figure 2: Control-flow structure

and therefore inexpensive optimization techniques. Consequently, extracting branch distance information separately for each hot spot delivers maximum performance benefits at a minimal hardware cost.

By utilizing application knowledge regarding the control structure of each hot spot and the branch distance of each BB, a specialized hardware structure can efficiently track the control-altering instructions in advance by simply adding the branch distance to the starting address of the corresponding BB. In our technique, this is performed by a *Branch Identification Unit* (BIU), with the detailed hardware implementation being elaborated in Section 5.

4.2 Filtering unnecessary BTB accesses

4.2.1 Low-power BTB access policy

The proposed early branch identification technique enables an extremely power-efficient BTB access policy, which performs BTB lookups only for instructions that need target addresses, that is, only for unconditional jumps and conditional branches predicted taken. To achieve this, in addition to the branch distance, the information regarding the type of the branch should also be stored in the BIU. During dynamic execution, if the obtained branch type information indicates that the subsequent control-altering instruction is a return instruction, only the return address stack is accessed. If it is an unconditional jump, only the BTB is accessed to obtain the target address.

In the case where the subsequent control-altering instruction is a conditional branch, our filtering strategy accesses the BTB only if the branch is predicted to be taken. The value of the branch distance determines whether or not an access to the dynamic branch predictor can be completed before initiating a subsequent BTB access. Consequently, to avoid incurring any additional latency in BTB accesses, different access policies should be selected for the BTB as well as the branch predictor according to the value of the branch distance. To be more concrete, suppose the branch distance is D, the number of instructions fetched per cycle is n, and the latency to access the BIU and calculate the next branch address is t cycles. Accordingly, the next branch address is available in t cycles, whereas the exact branch is brought into the processor frontend in D/n cycles, at which time the associated BTB access should be performed to ensure no performance degradation. Consequently, if the time interval $(D/n - t)$ is **longer** than the latency to access the dynamic predictor, a predictor access can be completed before starting a BTB access.

To guide BTB lookups when a predictor access cannot be completed before starting a BTB access, for each basic block a static prediction of the corresponding subsequent branch is also cached in the BIU. Consequently, if the time interval $D/n - t$ is **shorter** than the access latency of the dynamic predictor, a BTB lookup is conditionally performed based on the value of the static prediction. As the static prediction cannot achieve the same accuracy as the dynamic predictor, this access policy causes a slight degradation in performance. However, compared with the significant power reduction achieved by this access policy, the resulting slight performance degradation is negligible.

4.2.2 Handling single instruction basic blocks

In the aforementioned access policy, a BTB lookup can be performed as soon as the next branch address becomes available. Because the BIU is a small SRAM structure, typically the access to the BIU and the subsequent computation of the next branch address can be completed within 1 cycle. Since the BIU is accessed in parallel with the fetching of the first instruction of a basic block, initiating a BTB lookup after a BIU access imposes no performance degradation as long as the first instruction of that basic block is not a control-altering instruction. However, in the extreme case where a basic block is nothing but a single[2] control-altering instruction, a 1-cycle additional latency is incurred in the process of awaiting the calculation of the next branch address.

To handle this problem, in our framework basic blocks solely composed of a single control-altering instruction are encoded in their corresponding **predecessors**. In other words, for each basic block, two dedicated flags are employed to indicate whether either of its two subsequent BBs is composed solely of a single control-altering instruction. An illustrative example can be seen in the CFG presented in Figure 2b. If the basic block $B3$, which is the target BB of $B2$, only consists of instruction *jump1*, a flag is set for $B2$ to encode this information. During dynamic execution, if the pre-set flags obtained from a BIU access together with the predicted branch direction indicate that the incoming BB begins with a control-altering instruction, a BTB access rather than a BIU access is performed for the incoming basic block. This access policy introduces an additional benefit in that no BIU entry needs to be allocated for a basic block composed solely of a single control-altering instruction, thus reducing the total number of BIU entries necessitated for the corresponding hot spot.

4.3 Waking up hibernating predictor entries

Various techniques [7, 9] have been proposed for on-chip SRAM structures in order to identify inactive entries that

[2]Our statistical results indicate that only 4.9% on the average of all the dynamically executed branches lie on the first (and in this case are the only) instruction of a basic block.

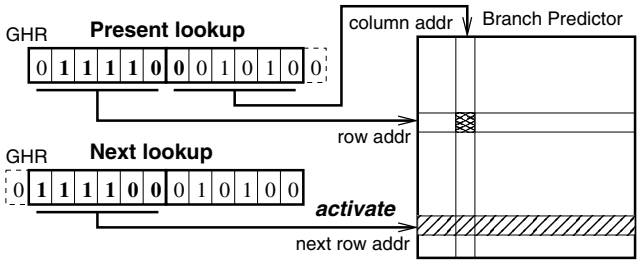


Figure 3: Activating BP rows indexed by GHR

are switched into hibernation mode to save leakage power. However, applying these techniques to branch predictors without adversely impacting performance is more complicated [8], as these techniques only focus on a turnoff mechanism while paying little attention to the activation of hibernating SRAM entries. More specifically, if an active entry is accidentally considered "idle" and switched into the hibernation mode, a significant performance penalty will be incurred for the next access. In order to overcome this performance degradation, in this paper we propose a **prefix analysis** technique to identify rows in the predictor that will be accessed relatively shortly.

The effectiveness of our technique can be attributed to the physical organization of branch predictors; although logically predictor tables are arrays of 2-bit counters, they are physically implemented as square or nearly-square array structures, in order to balance the complexity of decoders as well as the wordline and bitline delay. Since predictor entries are too small to be deactivated individually, a cost effective choice for turning off these counters is at the granularity of rows in the array structure, necessitating corresponding activation mechanisms of the same granularity. Because the row address constitutes the most significant part of the access index, designing an effective prefix analysis technique enables the identification and the pre-activation of the hibernating rows that will be accessed in short order. More specifically, prefixes can be generated in different ways according to the index mechanisms, detailed as follows.

- **Global history indices:** In this case, a single *global history register* (GHR) is employed to record the direction of the most recent n conditional branches, assuming the size of GHR is n bits. Accordingly, the next predictor access index is obtained by shifting the current GHR one bit to the left, with the last bit equal to the direction of the most recent branch. This implies that prefixes of the following predictor access indices can be easily extracted from the content of the current GHR, as shown in Figure 3. More specifically, the prefix of the next i^{th} predictor access consists of bit $(\lfloor n/2 \rfloor - i)$ to bit $(n-1-i)$ in the current GHR, implying that all the prefixes up to the next $(\lfloor n/2 \rfloor)^{th}$ branches can be extracted from the current content of the GHR, thus enabling the corresponding rows to be pre-activated as well.

- **Branch address indices:** If branch addresses are used as the predictor access index, the next access index is not necessarily identical to the current index. However, in most cases the next access prefix can still be identified in time to pre-activate a hibernating entry because of two reasons. As the proposed BIU can identify in advance the next branch address and thus the next predictor access prefix, the pre-activation latency can be hidden if the branch distance indicates that the next branch is not forthcoming. Furthermore, the small code size of a hot spot makes changes in the access prefix, which lies in the middle section of the branch address, infrequent. This implies that the present branch and the next branch will probably map to the same row or adjacent rows in the predictor. An additional case that may need to be considered is the case where the branch distance is small while the difference between the two consecutive branch addresses is large. While this may indeed result in a predictor row in hibernation not being pre-activated in time, the simulation results in Section 6 show this to be quite a rare occurrence.

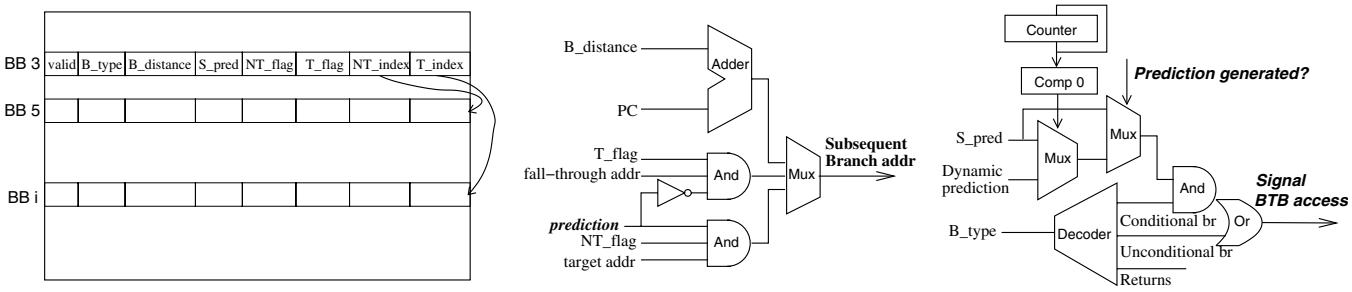
As most of the state-of-art predictors are indexed by linear functions of global histories and branch addresses, prefixes can be generated by combining the two individual prefix analysis results using the same function as the one used for generating the access index. For example, the index to the well-known *gshare* predictor [13] is generated by *xor*ing the global history and the corresponding branch address, implying the next access prefix can be generated by *xor*ing the prefixes of the next global history and the next branch address, both of which can be analyzed as described above.

The proposed prefix analysis technique can cooperate with all the circuit-level leakage power reduction techniques that maintain a hibernation mode in addition to the normal functional mode. In this paper, we employ the circuit-level technique originally proposed in [9], which dynamically modulates the power supply voltage to reduce the leakage current, thus preserving the original state of an SRAM cell. As this circuit-level technique only needs 1 additional cycle to awaken a hibernating cell, rows in the branch predictor can be periodically switched into hibernation aggressively to achieve additional power savings. More specifically, because no overlapping of code exists between different hot spots, during dynamic execution, all the rows in the branch predictor can be switched into hibernation before entering a hot spot. As the hot spot starts to be executed, the proposed prefix analysis technique is used to identify and pre-activate hibernating rows for subsequent accesses. In the worst case where a hibernating row can not be activated in time, the static prediction stored in the BIU is used to guide the subsequent BTB access and hence the next instruction fetch. Because of the strong temporal locality associated with each hot spot, fewer pre-activations suffice during execution, since more incoming branches will probably have been executed. Once all the predictor rows in the current working set have been activated after several iterations, no pre-activation needs to be performed until the execution flow switches to another hot spot.

5. IMPLEMENTATION

5.1 Constructing the BIU

According to the analysis in the last section, for each basic block in a hot spot, four pieces of information need to be



(a) BIU organization

(b) Generating next branch address

(c) Signaling BTB accesses

Figure 4: Hardware implementation

extracted from profiling: branch type, branch distance, static prediction, and two flags used to indicate whether either of the two subsequent BBs following the next branch constitutes a single control-altering instruction. The proposed BIU table is designed to contain one entry per BB to store all the necessary information. As can be seen in Figure 4, each BIU entry includes a *valid* bit, a *B_type* field, a *B_distance* field, an *S_pred* bit, as well as the aforementioned two flags, *NT_flag* and *T_flag*. Furthermore, to efficiently access the BIU with no need of associative lookups and tag comparisons, two additional fields, *NT_index* and *T_index*, are used to record the BIU indices corresponding to the fall-through BB and the target BB following the next branch, respectively.

Transferring the extracted information of each hot spot to the BIU is performed by having the BIU table accessible by software. More specifically, prior to entering any application hot spot, a sequence of instructions inserted by the compiler is executed to store the extracted information within the BIU. The introduced performance overhead is practically nonexistent since the setup code is executed only once prior to entering a hot spot. After completing the setup step, all the basic blocks of a hot spot are mapped to the BIU in linear order in the program code. When the hot spot commences execution, the first entry in the BIU, that is, the entry corresponding to the starting BB, is accessed immediately. After completing a BIU access, the index for the next BIU access equals either the *NT_index* field or the *T_index* field in the current BIU entry, to be selected by the direction of the incoming branch.

The organization presented in Figure 4 indicates that our BIU is an extremely power efficient structure. As the number of basic blocks in a hot spot is relatively small, in most cases the required BIU size can be limited to within 128 entries. Furthermore, the width of the BIU entries is also tiny. The valid bit, the *S_pred* field, and the two flags are all 1 bit wide. The *B_type* field is 2 bits wide. The index fields are 7 bits wide, assuming a BIU size of 128 entries. The *B_distance* field can be implemented in 5 bits, since typically the branch distance does not exceed 32 instructions. In total, each entry is only 25 bits wide. Since the size of the BIU is much smaller than the instruction cache, the latency to access the BIU and calculate the next branch address can be effectively hidden and introduces no pipeline timing constraints even for deeply pipelined front-ends. Furthermore, as a dedicated BIU entry is allocated per BB, no conflicts or misses can exist in our BIU, implying that the early identification of the next branch address is always guaranteed.

5.2 Generating the next branch address

The generation of the next branch address is performed as soon as the branch distance information is obtained through a BIU access, in parallel with the fetching of the initial instruction of the corresponding BB from the instruction cache. In the typical case, the next branch address for the present BB equals the sum of the *B_distance field* and the program counter. Moreover, if the incoming BB begins with a control-altering instruction, an additional BTB access needs to be performed for the incoming BB. The various cases can be detailed as follows:

1. *next branch (branch $\mathcal{A}$) addr* for **present** BB
   $\Leftarrow$ *B_distance + PC*
2. **if** *T_flag* field == 1 **&&** branch $\mathcal{A}$ == taken
   next branch (branch $\mathcal{B}$) addr for **incoming** BB
   $\Leftarrow$ fall-through addr of branch $\mathcal{A}$
3. **if** *NT_flag* field == 1 **&&** branch $\mathcal{A}$ == not-taken
   next branch (branch $\mathcal{B}$) addr for **incoming** BB
   $\Leftarrow$ target addr of branch $\mathcal{A}$

Figure 4b presents the hardware logic to implement these three cases. In most situations only one BIU access suffices for the calculation of the next branch address. The only exception occurs in the case where the corresponding branch distance exceeds the representation capability of the *B_distance* field, forcing the allocation of multiple entries in the BIU and the execution of multiple lookups to obtain a large branch distance. While performing multiple lookups implies more power consumption, no performance degradation is introduced here, since the corresponding large branch distance indicates that the next branch is at least more than 2^n instructions away, assuming an n-bit width of the *B_distance* field.

5.3 Signaling BTB accesses

As discussed in Section 4.2, the BTB is accessed for unconditional jumps and conditional branches predicted to be taken. According to the value of the *B_distance* field, a conditional branch is predicted either by the *S_pred* field stored in the BIU or by the dynamic branch predictor. This access policy can be summarized as follows:

1. Access the BTB **if** (*B_type* == unconditional)
   || (*B_type* == conditional **&&** *pred* == taken)

	Branch%	Taken%	4K*bimod* hit%	BTB hit%	4K*gshare* hit%	BTB hit%
adpcm_e	28.68	21.22	70.79	70.78	84.20	84.19
epic	14.71	8.08	95.69	95.68	96.16	96.15
gsm_e	4.86	4.05	93.33	93.24	93.65	93.56
meg2_e	17.02	10.38	76.91	76.90	81.58	81.58
mcf	21.10	13.45	91.84	91.84	95.44	95.43
gzip	12.05	7.62	92.78	92.78	94.15	94.15
gcc	13.12	8.53	94.53	92.38	94.54	92.39
parser	15.48	10.05	91.83	91.72	95.57	95.47
twolf	12.08	6.93	87.16	87.16	86.70	86.70
gap	12.65	9.18	96.12	92.88	97.88	94.64
vpr_route	10.65	5.50	94.05	94.03	94.11	94.09
average	14.76	9.54	89.55	89.04	92.18	91.67

Table 1: Branch characteristics of the benchmarks

2. **if** $(D/n - L_{BIU} < L_{BP}) \parallel (D/n - L_{BIU} < L_{BP} + L_{act}$ **&&** BP entry is in hibernation)
   pred $\Leftarrow$ *S_pred*
   else *pred* $\Leftarrow$ dynamic prediction result

wherein D represents the value of the *B_distance* field, n the number of instructions fetched per cycle, L_{BIU} the latency to access the BIU and compute the next branch address, L_{BP} the latency to access an active BP entry, and L_{act} the latency to activate a hibernating BP entry.

The implementation of this access policy is presented in Figure 4c. After a BIU access has been performed, the counter register is loaded with the value $D/n-t$. The *B_type* field is forwarded to a decoder and a BTB access is directly signaled for unconditional jumps. In the case of conditional branches, if no dynamic prediction has been generated either before the counter turns 0 (because of a short branch distance) or even after the counter turns 0 (because the corresponding predictor row is in hibernation), a BTB access is performed if the *S_pred* field equals 1. Otherwise, the BTB access will be conditionally signaled by the dynamic predictor.

6. SIMULATION RESULTS

To evaluate the proposed power reduction techniques for different types of applications, a set of experimental studies have been performed on both the *Mediabench* [14] and the *SPECint 2000* benchmarks. *ATOM* [15] is used to instrument the assembly code to identify the hot spots and to analyze the corresponding control structures. The *sim-bpred* simulator of the *SimpleScalar* toolset [16] is modified to simulate the behavior of the proposed branch subsystem. Such a simulation environment corresponds to an in-order pipelined processor, which constitutes a typical case for embedded processors with stringent power constraints.

Table 1 lists the branch characteristics of the benchmarks running on a baseline architecture with a configuration consisting of a 256-set 4-way associative BTB and a 4K branch predictor. The first column, denoted as *Branch%*, lists the ratio of branches to the total number of executed instructions, a numerical characteristic of the branch instruction density. It can be observed that the branch density for most benchmarks is consistently in the range of 10% to 25%, a typical situation for real applications. The significantly low branch density of 4.86% for the *gsm_e* is due to the existence of a basic block containing more than 300 instructions. As previous BTB filtering techniques [1, 2, 3] have to access the BTB for all branch instructions, these branch density values actually reflect the upper bound of power reduction that can be achieved by previous work. The second column, denoted as *Taken%*, lists the percentage of taken branches out of the total number of executed instructions. The next two columns present the prediction accuracy of a 4K *bimod* predictor[3] and the corresponding hit rate of the 256-set 4-way BTB. In a perfect situation with no BTB conflicts, the BTB hit rate should be equal to the branch prediction accuracy. In practice, however, a difference exists which is due to branches correctly predicted by the predictor, yet not found in the BTB when attempting to obtain their target addresses. As can be observed from Table 1, the differences between the BTB hit rate and the branch prediction accuracy are negligible except for *gsm_e, gcc, parser*, and *gap*, implying that a 256-set 4-way BTB suffices to capture almost all of the correctly predicted branches in the other benchmarks. To illustrate the influence of different indexing mechanisms, similar results accumulated for a 4K *gshare* predictor are presented in the last two columns. As it can be seen, the *gshare* predictor can reach a higher prediction accuracy for most benchmarks, since it enables the exploitation of correlation between multiple branches.

6.1 Power results

We utilize *CACTI* [17] to analyze the access power characteristics of the BIU, the BTB, and the branch predictor, since all of them are implemented as standard SRAM structures. More specifically, the BTB is modeled as a standard cache structure, while the power characteristics of accessing the branch predictor and the proposed BIU are extracted through modeling a direct-mapped cache structure and subtracting the power consumption of the tag array and the comparator cells. The line width of both the BTB and the proposed BIU are fixed at 8 bytes, the minimal word size allowed by CACTI. Moreover, since our framework employs the circuit-level technique originally proposed in [9] to reduce leakage power, we also use the leakage power characteristics reported in [9]. For each SRAM cell, these power characteristics include the leakage energy consumed in both the normal mode and the hibernation mode, as well as the dynamic power consumed to switch between these two modes.

[3]A *bimod* predictor uses pure branch addresses as the access indices.

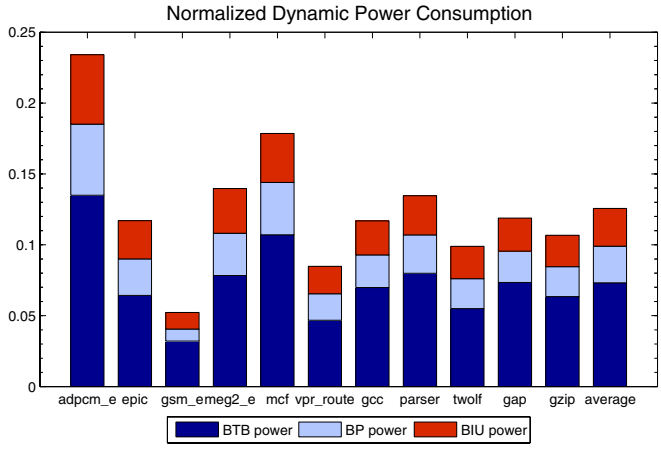


Figure 5: Normalized dynamic power consumption for the BTB and the predictor

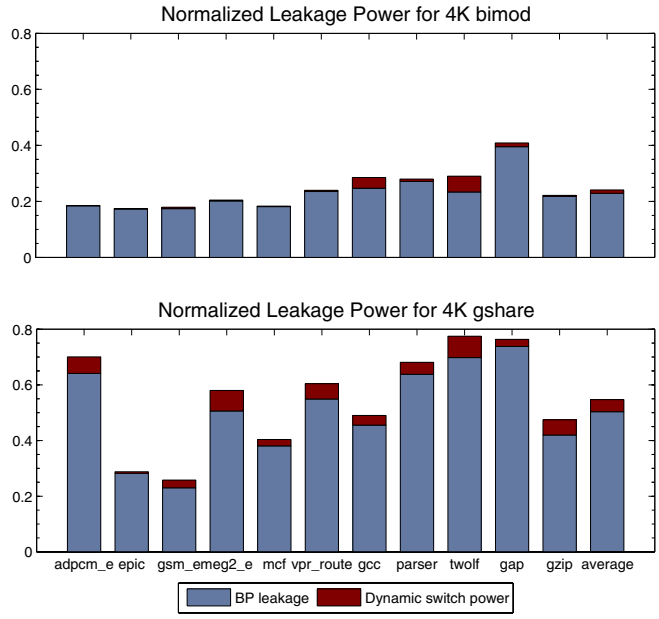


Figure 6: Normalized leakage power consumption for the branch predictor

The obtained dynamic and static power characteristics, together with the number of accesses and the number of hibernation cycles reported by the modified SimpleScalar simulator, are used to evaluate the total energy consumption. As both leakage and dynamic power values vary heavily with different designs and fabrication processes, in this paper we focus for evaluation purposes on the ratios of values. In addition, since the proposed methodology is independent of the rest of the processor architecture, we only report power savings results on the branch subsystem. A measure of the importance of power consumption in a branch subsystem can be observed by noting that a number of researchers [1, 6] have estimated the contribution of the branch subsystem to the total power consumption to exceed 10%.

Reduction in dynamic power: Figure 5 reports the dynamic power consumption of the proposed branch prediction subsystem, normalized to a traditional branch subsystem that accesses the BTB and the branch predictor for each instruction under fetch. The presented data is accumulated for a simulation configuration consisting of a 128-entry BIU, a 256-set 4-way associative BTB, and a 4K *gshare* predictor. For each benchmark, the data reported here corresponds to the entire program, including both the hot spots and the remaining parts. Each power consumption value is broken down into the power consumed by the BTB, the BIU, and the branch predictor, in order to illustrate the contribution of each individual component. As it can be seen, the proposed technique achieves a significant reduction in dynamic power consumption, ranging from 76.6% to 94.8%, with the average equaling 87.4%. Not surprisingly, more power savings can be achieved if the benchmark displays a relatively lower branch density. In addition, the extra power spent in accessing the BIU for each basic block is much less than the significant power reduction achieved through filtering the BTB accesses, implying that the dynamic power overhead introduced in our technique is negligible. More importantly, for each benchmark except *gsm_e*, the value presented in Figure 5 is always smaller than the value listed in the first column of Table 1, implying that the extra energy savings achieved through filtering the BTB accesses for *not-taken* branches exceeds the energy overhead introduced by the BIU. The only exception is *gsm_e*, in which only 16% of all the dynamic executed instructions are resolved to be *not-taken*. As the first column of Table 1 constitutes a lower bound of dynamic power consumption for previous BTB filtering techniques, these values indicate that the proposed technique outperforms all previous BTB filtering techniques in reducing dynamic power.

Reduction in leakage power: Figure 6 reports the normalized leakage power consumed by the branch predictor in applying the turn-off and wake-up policies outlined in Section 4.3. Both leakage power savings and dynamic power spent in activating hibernating predictor entries are taken into consideration. To illustrate the influence of different indexing mechanisms, the results for both a 4K *bimod* and a 4K *gshare* predictor are presented. According to the leakage power characteristics reported in [9], we assume that for each SRAM cell the leakage power consumed in the hibernation mode is 16% of the normal mode, while the dynamic switch power is 10 times of the original leakage power. As it can be seen, the leakage power consumed by the *bimod* predictor can be reduced to 22.9% on average, which is quite close to the lower bound of 16%. In fact, most predictor rows are kept in hibernation during execution, as the *active ratio* (the percentage of active predictor rows) for the *bimod* predictor is only 8.4% on average. This is because the *bimod* predictor displays a strong temporal locality since each static branch only touches one predictor entry, as well as a strong spatial locality since conditional branches within a neighborhood conceivably are mapped to the same or consecutive predictor rows during execution. This strong locality also results in a limited number of pre-activations required by the *bimod* predictor; only 1.2% of the original leakage power is spent in activating hibernating predictor entries. On the other hand, because each static branch can be mapped to multi-

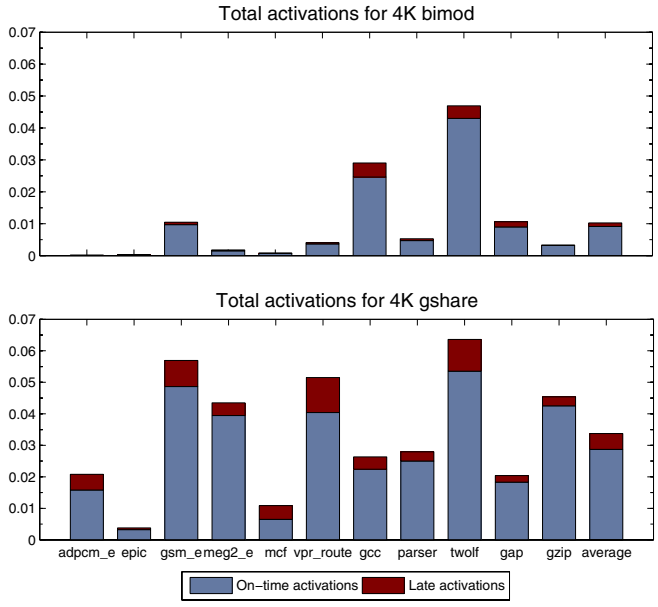


Figure 7: Pre-activations needed for accessing predictor entries in hibernation

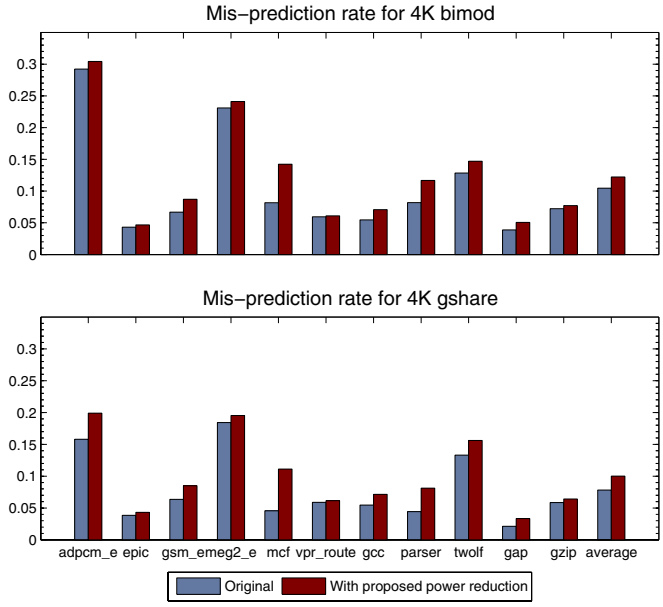


Figure 8: Impact on mis-prediction rate

ple entries in the *gshare* predictor, the corresponding average active ratio is 41.0%, almost 5 times that of the *bimod* predictor. However, even for the *gshare* predictor, the leakage power can still be significantly reduced to 50.3% on average, and only 4.4% of the original leakage power is spent in activating hibernating rows. This indicates that even for predictor structures designed to hash branch addresses over many entries, the proposed turn-off and wake-up policies still show significant promise for addressing leakage concerns. In sum, the proposed leakage power reduction technique is quite effective, as it can achieve a 75.9% reduction in leakage power for a 4K *bimod* predictor, and a 45.3% reduction for a 4K *gshare* predictor.

6.2 Performance results

As discussed in Section 4.2, the latency to access the BIU and calculate the next branch address can be completely hidden in the proposed framework. As a result, the proposed power reduction technique only affects the system's overall performance by reducing prediction accuracy. This occurs when no dynamic prediction can be obtained before the corresponding branch instruction is brought into the front-end, resulting in the *S_pred* field stored in the BIU being used to guide the BTB access and subsequent instruction fetch. In the proposed framework, the inability to complete an access to the dynamic predictor is attributed to **either** a short branch distance **or** the hibernation of the corresponding predictor row. Fortunately, the proposed prefix analysis technique can effectively hide the penalty caused by the second case.

Effectiveness of pre-activation: To evaluate the effectiveness of the proposed prefix analysis technique, Figure 7 presents the percentage of executed branches that need to be pre-activated for the 4K *bimod* and *gshare* predictors. We do **not** report the results of pure global history indices, since in that case prefixes for subsequent predictor accesses can **always** be identified in advance, ensuring consistent on-time pre-activations. Each value reported in Figure 7 is broken down into *on-time pre-activations* and *late activations*. It can be observed from Figure 7 that the proposed pre-activation policy is quite effective, as it can capture on average 89.0% of all the accesses to hibernating rows for the *bimod* predictor and 85.1% for the *gshare* predictor. Here, the *bimod* predictor again outperforms the *gshare* predictor in that fewer pre-activations are needed and an increased percentage of pre-activations can be performed on time. In fact, the *bimod* predictor displays quite a strong temporal and spatial locality, as for most benchmarks the number of executed branches that need to be pre-activated is less than 1%. While this may limit the importance of our pre-activation technique for the *bimod* predictor, the more accurate *gshare* predictor still has a non-negligible number of branch instructions mapped to hibernating predictor entries, thus requiring the use of the proposed prefix analysis technique to effectively hide the activation latency.

Increase in mis-prediction rate: Figure 8 presents the increase in mis-prediction rate caused by the proposed framework, with the proposed prefix analysis technique incorporated to hide pre-activation latency. As can be observed, for *adpm_e, mcf* and *parser* which display a relatively larger branch density, the *S_pred* field is used more frequently, resulting in a relatively higher degradation in prediction accuracy. On average a 1.8% increase in mis-prediction rate is obtained for the *bimod* predictor, and similarly a 2.2% increase for the *gshare* predictor. While the increase in the mis-prediction rate seems non-negligible at first sight, the fundamental reason is that we employ a simple approach of predicting all forward branches to be not-taken and all backward branches to be taken for statically setting the *S_pred*. As can be seen, even with such an inaccurate static prediction scheme, the proposed prefix

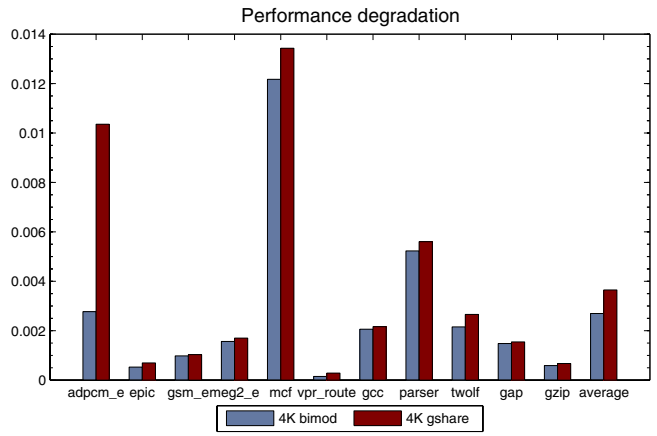


Figure 9: Performance degradation

analysis technique still effectively pre-activates hibernating predictor entries, thus significantly reducing the usage frequency of the *S_pred* field. This degradation in prediction accuracy can be easily reduced if each static branch is guided more precisely through static profiling results.

Performance degradation: Figure 9 shows the performance degradation caused by using the *S_pred* field stored in the BIU when no dynamic prediction can be obtained before the corresponding branch instruction is brought into the front-end. In can be observed that the performance degradation caused by the proposed technique is negligible, ranging from 0.02% to 1.34%, with the average degradation of 4K *bimod* and *gshare* predictors respectively equal to 0.27% and 0.37%. Throughout all the benchmarks the *gshare* predictor displays a higher degradation in performance, since the original prediction accuracy associated with the *gshare* predictor is consistently higher than the *bimod* predictor. Not surprisingly, the data presented here is strongly correlated to the mis-prediction rate presented in Figure 8 in that a relatively higher performance degradation occurs for *adpm_e, mcf* and *parser* wherein the *S_pred* field is used more frequently due to their larger branch density.

7. CONCLUSIONS

We have presented a methodology to design a power-efficient branch subsystem in this paper. A *branch identification unit* (BIU) has been proposed to achieve early identification of incoming branch addresses according to the static extracted program information regarding the control-flow structure as well as the branch distance. By accessing the cost-efficient BIU once per basic block, the highly power expensive cycle-by-cycle BTB lookups are replaced by BTB accesses performed only for branches predicted taken. Furthermore, hibernating predictor rows can be pre-activated for subsequent accesses, minimizing the performance degradation originally imposed by applying leakage power reduction techniques to branch predictors. The proposed framework enables the integration of the traditional branch prediction subsystem methodology into a wide range of embedded processor architectures.

8. REFERENCES

[1] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. *in Proc. 8th HPCA*, pages 233–244, Feb. 2002.

[2] P. Petrov and A. Orailoglu. Low-power branch target buffer for application-specific embedded processors. *IEE Transactions on Computers & Digital Techniques*, 152(4):482–488, July 2005.

[3] M. Monchiero, G. Palermo, M. Sami, C. Silvano, V. Zaccaria, and R. Zafalon. Low-power branch prediction techniques for VLIW architectures: A compiler-hints based approach. *Integration, the VLSI Journal*, 38(3):515–524, Jan. 2005.

[4] A. Baniasadi and A. Moshovos. Branch predictor prediction: a power-aware branch predictor for high-performance processors. *in Proc. ICCD: VLSI in Computers and Processors*, pages 458–461, Sep. 2002.

[5] A. Baniasadi and A. Moshovos. SEPAS: A highly accurate energy-efficient branch predictor. *in Proc. ISLPED '04*, pages 38–43, Aug. 2004.

[6] D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. C. Huang. Branch prediction on demand: an energy-efficient solution. *in Proc. ISLPED '03*, pages 390–395, Aug. 2003.

[7] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behaviour to reduce cache leakage power. *in Proc. 28th ISCA*, pages 240–251, July 2001.

[8] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. *in Proc. ICCD: VLSI in Computers and Processors*, pages 442–445, Sep. 2002.

[9] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. *in Proc. 29th ISCA*, pages 148–157, May 2002.

[10] P. Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. *in Proc. 28th Micro*, pages 252–257, Dec. 1995.

[11] J. E. Smith. A study of branch prediction strategies. *in Proc. 8th ISCA*, pages 135–148, May 1981.

[12] S. T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *in Proc. 5th ASPLOS*, pages 76–84, Oct. 1992.

[13] S. McFarling. Combining branch predictors. *Tech. Note TN-36, DEC WRL*, June 1993.

[14] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *in Proc. 30th Micro*, pages 330–335, Dec. 1997.

[15] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *Tech. report, Western Research Lab*, March 1994.

[16] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[17] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. *Tech. report, Western Research Lab*, Aug. 2001.