Low Power/Area Branch Prediction Using Complementary Branch Predictors

Resit Sendag[1], Joshua J. Yi[2], Peng-fei Chuang[3], and David J. Lilja[3]

[1] – Electrical and Computer Engineering
University of Rhode Island
Kingston, Rhode Island

[2] – Networking and Multimedia Group
Freescale Semiconductor, Inc.
Austin, The Great State of Texas

[3] – Electrical and Computer Engineering
University of Minnesota – Twin Cities
Minneapolis, Minnesota

Abstract

Although high branch prediction accuracy is necessary for high performance, it typically comes at the cost of larger predictor tables and/or more complex prediction algorithms. Unfortunately, large predictor tables and complex algorithms require more chip area and have higher power consumption, which precludes their use in embedded processors. As an alternative to large, complex branch predictors, in this paper, we investigate adding *complementary branch predictors* (CBP) to embedded processors to reduce their power consumption and/or improve their branch prediction accuracy. A CBP differs from a conventional branch predictor in that it focuses only on frequently mispredicted branches while letting the conventional branch predictor predict the more predictable ones. Our results show that adding a small 16-entry (28 byte) CBP reduces the branch misprediction rate of static, bimodal, and gshare branch predictors by an average of 51.0%, 42.5%, and 39.8%, respectively, across 38 SPEC 2000 and MiBench benchmarks. Furthermore, a 256-entry CBP improves the energy-efficiency of the branch predictor and processor up to 97.8% and 23.6%, respectively. Finally, in addition to being very energy-efficient, a CBP can also improve the processor performance and, due to its simplicity, can be easily added to the pipeline of any processor.

1. Introduction

High branch prediction accuracy is a necessary component for high performance in today's processors. Processor designers typically increase the branch predictor's prediction accuracy by using more complex algorithms and larger prediction tables. Unfortunately, this approach is difficult to implement in embedded processors for several reasons. First, due to longer training times, higher prediction latencies, and a higher misprediction penalty, a more complex and/or larger branch predictor may actually result in a net performance loss, despite its higher prediction accuracy [12]. Second, the larger chip areas and higher power consumption of larger and more complex branch predictors may make them too large and/or power hungry. Finally, for reasons including implementation difficulty, cost, and design time, processor designers are unlikely to significantly increase the size of the branch predictors in their current-generation processors, much less replace them altogether, *e.g.*, replacing a static branch predictor with a dynamic one.

To favorably address the trade-off between the branch predictor's performance and power/area, in this paper, we propose adding *complementary branch predictors* [19] to the *conventional* (existing) branch predictors of embedded processors. The key difference in how complementary and conventional branch predictors make branch predictions is that complementary branch predictors only make predictions for the subset of branches that degrade the processor's performance, namely, frequently mispredicted branches. Therefore, instead of making branch predictions for all branch instructions, which is very costly from a power consumption point-of-view and puts the branch predictor on the critical path of the pipeline (thus increasing the branch misprediction penalty), complementary branch predictors complement the conventional branch predictor by only making predictions for the branches that the branch predictor has trouble predicting accurately. Complementary branch predictors are based on the fact that patterns of branch mispredictions exist for all branch predictors, and can be detected and exploited to improve the processor's branch predictor accuracy.

To quantify the efficacy of this approach, we implemented the branch misprediction predictor

(BMP). This mechanism uses the branch misprediction history to predict which future branch will mispredict next and when that will occur. Then, before the misprediction actually occurs, the BMP changes the prediction to avoid a misprediction (and the subsequent recovery) so the processor can continue executing down the correct-path. Since it only focuses on the mispredicted branches, it can improve the branch prediction accuracy of any branch predictor, static or dynamic, simple or complex. Also, the BMP can be added to any conventional branch predictor. Therefore, it offers a different, more efficient approach of partitioning the branch predictor's hardware budget. Furthermore, since they target future branches only, it is not on the processor's critical path.

This paper makes the following contributions:

1. We propose adding complementary branch predictors to: A) Improve the performance or energy-efficiency and B) Reduce the chip area (cost) of embedded processors.
2. We show how complementary branch predictors can be designed and implemented with the branch predictors commonly found in embedded processors.
3. We show that complementary branch predictors can significantly improve the branch prediction accuracy of the branch predictors in embedded processors for SPEC 2000 and MiBench benchmarks, and can significantly improve the IPC and EDP of those processors, as compared to larger conventional branch predictors.

The remainder of this paper is organized as follows: Section 2 describes the implementation the BMP in more detail. Section 3 presents a code example where the branch predictor fails, but the BMP correctly predicts the branch direction. Section 4 describes the evaluation methodology and presents the BMP performance (prediction accuracy, power consumption reduction, and EDP) results. Finally, Section 5 presents some related work and Section 6 concludes.

2. Complementary Branch Prediction: The Branch Misprediction Predictor

2.1. Description

Our BMP uses a simple mechanism to detect branch misprediction patterns. Namely, it counts the number of committed branches (the distance) between consecutive branch mispredictions for that particular context (*i.e.*, index). The distance represents a prediction as to when the next branch misprediction will occur.

2.2. Implementation and Operation

The main component of the BMP is the MPBT, or the mispredicted branch table, which is shown in Figure 1. The width of each MPBT entry is 14 bits wide; 4 bits for the PC, 8 bits for the distance, 1 used bit, and 1 prediction direction (taken/not taken) bit. We form the index to the MPBT by XOR-ing the folded PC (*i.e.*, bits 2 to 9 of the PC XOR-ed with bits 10 to 17 of the PC) of current mispredicted branch with a concatenation of the global history bits (GHist) and global misprediction history bits (GMPHist), and with the branch misprediction distance (BCR), which is the number of branches between the last two mispredictions.

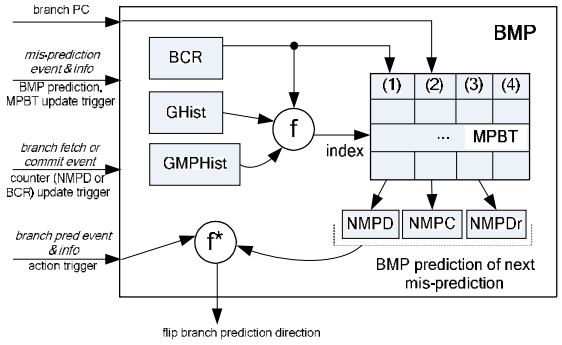

Figure 1. BMP components and operation. For simplicity, this figure only shows major components. *f* is logic that generates the MPBT index while *f** intervenes to correct potential mispredictions. Note that the MPBT is only accessed and updated after a mispredicted branch, and the BCR and the NMPD are updated on committed and fetched branches, respectively.

Misprediction Prediction: After a branch misprediction, the BMP uses the index to access the MPBT and the corresponding entry is copied into: 1) The 8-bit misprediction distance (NMPD) register and a 2) 5-bit register that holds the 4-bit next-to-be-mispredicted PC (NMPC) and a 1-bit T/NT field (NMPDr).

For every branch that is fetched, the NMPD decrements. For every committed branch, the BCR increments. When a misprediction occurs, the BCR register is reset and the BMP also copies new values into the NMPD, NMPC, and NMPDr registers. When the NMPD decrements to zero, the BMP predicts that the next branch instruction will be mispredicted and corrects the predicted direction only if the: 1) bits 3 to 6 of the PC match the NMPC field and 2) Branch predictor's predicted direction is the same as the NMPDr register.

It is important to note that the output of the BMP is a prediction of the distance and address of the next-to-be-mispredicted branch. This output is fundamentally different than the predicted direction that is the output of conventional branch predictors (*i.e.*, a direction).

Updating the MPBT: To track the number of correct branches between mispredictions, the BMP uses an 8-bit branch counter, the BCR, which increments each time a branch commits. After a branch misprediction, the BMP updates the 1) 8-bit NMPD field with the value of the BCR register, 2) 4-bit NMPC field with bits 3 to 6 of the misprediction branch's PC, and 3) 1-bit NMPDr field with the predicted direction (which was wrong) for the corresponding MPBT entry.

Evicting MPBT Entries: A BMP prediction is considered to be correct when the NMPC and NMPDr fields match the PC and predicted direction, respectively, after the NMPD register decrements to zero. Correct predictions set the used bit for that entry. However, to protect the MPBT from evictions based on a single BMP misprediction and/or aliasing, it takes two incorrect predictions to evict an MPBT entry; the first incorrect prediction clears the used bit only, while second clears all fields.

Power Efficiency of the BMP: Since 1) the BMP is essentially dormant for most of time, 2) the BMP is small, and 3) Small BMPs can still significantly improve the branch prediction accuracy, the BMP is both extremely energy and power efficient. Section 4.3 evaluates the energy-efficiency of the BMP.

3. Analysis of Why the BMP Works

In this section, we analyze why misprediction patterns occur and how the BMP helps to correct future mispredictions. Section 3.1 investigates whether or not branch prediction table conflicts account for the majority of the mispredictions that the BMP corrects. In Section 3.2, we discuss a synthetic code example to show why misprediction distance is a good metric for predicting future branch mispredictions, why conventional predictors fail, and BMP succeeds. And, finally, in Section 3.3, we give two code snippets from SPEC INT benchmarks, where the BMP works well.

3.1. Branch Predictor Conflicts

To investigate why and where a BMP helps to correct branch mispredictions, we first check whether the BMP primarily corrects mispredictions that are the result of conflicts in branch predictor table, *i.e.*, aliasing. Conflicts occur when multiple branch-history pairs share the same location in the branch predictor table. Figure 2 shows percentage of branch mispredictions corrected by BMP that are due to the conflicts for varying sizes of *gshare* branch predictor. This figure shows the average behavior of 8 selected SPEC benchmarks (*gcc, eon, perlbmk, gap, vortex, mesa, fma3d,* and *apsi*) where the BMP does very well. To filter out the impact of fixed loop counts, benchmarks where both BMP and a loop predictor do well are not included.

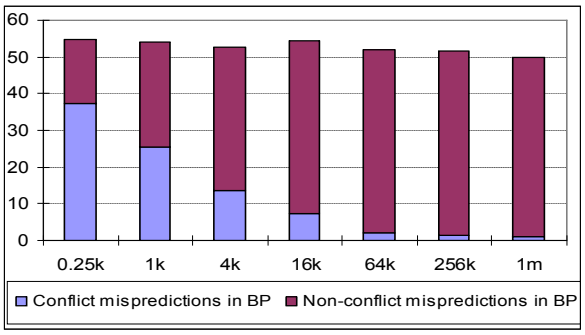

Figure 2. Percentage reduction in the branch misprediction rate with a 0.5KB BMP for varying sizes of a gshare branch predictor

Figure 2 shows that, for a constant 0.5KB-sized BMP, the percentage of mispredictions corrected by BMP that are due to conflicts in the branch predictor table decreases as the branch predictor size increases, from 60% when using 0.25KB branch predictor to less than 4% for 64KB and larger branch predictors. Since the percentage of the mispredictions due to conflicts decreases dramatically for increasing branch predictor sizes (From 68% for a 0.25KB *gshare* to 1.8% for a 1MB *gshare* (not shown)) and since the BMP can reduce the overall branch misprediction rate by about 50% for different sizes of the *gshare* predictor, as shown in Figure 2, we conclude that the BMP does not primarily correct mispredictions that are due to conflicts in the branch predictor tables, but corrects mispredictions due to other non-capacity-based reasons.

3.2. A Synthetic, Representative Code Example

To understand the causes of branch mispredictions that are not due to conflicts, we analyze profile data and the source code for 8 SPEC 2000 benchmarks. We observed that in these benchmarks, 30% to 60% of the mispredictions that are corrected by the BMP are due to loop branches that have *varying* loop counts, which are longer than what a branch predictor can distinguish, or have early loop exits, such as a break in a `for` or a `while` loop. Figure 3 presents a synthetic, but representative, code snippet. Variations of this example code occur in all benchmarks (see Figure 5), often with unstable loop counts or early loop exits. This example

```
....
while (true) {
    N = 150;
    for (i = 0; i < N; i++)
      { /* This is a loop with
        alternating loop counts */ }
    N = 250 – N;
}
....
```

Figure 3. A code example where BMP works well.

```
1: for (p = table[hash]; p; p = next)
{
2:    next = p->next_same_hash;

3:    if (GET_CODE (p->exp) != REG
4:        || REGNO (p->exp)
          >= FIRST_PSEUDO_REGISTER)
5:     continue;
...
6:    remove_from_table (p, hash);
}
```

Figure 4. Snippet from *gcc*. Line 3 has a high branch misprediction rate

Figure 5. Code snippet of `memset`() from glibc,

```
   memset (dstp, c, len) {
   /* fills the first len bytes of the
memory area pointed to by dstp with the
constant byte c
    */
    /* setup codes */

1:  xlen = len / 8;
2:  while (xlen > 0)
    {
      /* fill the memory pointed by dstp 8-
byte by 8-byte
       */
3:    dstp += 8;
4:    xlen -= 1;
    }

    /* fill rest of bytes */
  }
```

which is used extensively by *vortex*. Line 2 has high a branch misprediction rate because the number of loop iterations of this loop depends on len.

shows a simple loop whose loop count alternates between 100 and 150. The exit branch for the `for` loop will be mispredicted for as many times as the `while` loop condition is true. In the example, 150 bits of history is needed to eliminate the mispredictions at the loop exit. Simulations confirm that various branch predictors with a 256KB hardware budget and an 8KB loop predictor mispredict the loop exit branch every time. By contrast, a 4-entry BMP can easily correct all of these mispredictions by predicting the next misprediction distance, which is either 101 or 151.

Another example where the BMP works is an early exit branch inside a `for` loop, which further complicates the branch history. This type of behavior is also often seen in the benchmarks that we studied. The early exit branch inside the `for` loop will be mispredicted often when it is taken. In the same manner that the BMP corrected mispredictions for the for loop in Figure 3, BMP can also correct this type of misprediction, while the other branch predictors that we tested do not.

In summary, the code in Figure 3 shows that a BMP is an alternative approach to exploit long branch histories. While some advanced branch predictors have been proposed, such as neural predictors [12] or the O-GEHL predictor [20], they are much more complex and larger than a simple BMP, and therefore they are not suited for embedded processors. The BMP complements conventional branch predictors to exploit very long histories without increasing the hardware complexity and delay.

3.3. Two Code Examples from SPECint 2000

In this section, we show two examples where the BMP complements the branch predictors. The first example is the pseudo code from the *gcc* benchmark as shown in Figure 4. At the register transfer language (RTL) optimization stage, *gcc* uses a chained hash table to store all the RTL expressions within a basic block (the RTL expressions between two labels, which is not necessarily the same as the basic blocks found in a later compilation pass) and applies common sub-expression elimination (CSE). Each bucket of the hash table is a linked list of RTL expressions that have the same hash code. During the CSE process, the hash table and its linked lists are traversed several times, in order to ensure that the expressions whose values are changed by storing a new expression are properly handled.

Execution profiles show that the code in line 3 causes many branch mispredictions, even though the same linked list or the sequence of linked lists is traversed repeatedly. One of the reasons is that the number of branches is too large to be captured by global history. Another reason is because of the varying linked list data. Our results show that using the pattern of mispredictions in conjunction with the global branch history can effectively correct the mispredictions in line 3.

The next example illustrates when a BMP captures the behavior of loops with varying loop trip counts that

branch predictors mispredict. Most loops have varying loop trip counts. For example, memory manipulation functions such as `memset()` or `memcpy()` contain a main loop that is bounded by the size of memory area to be modified (See Figure 5). *vortex* uses these two memory manipulation functions extensively. Traditional branch predictors perform very poorly in determining when the loop will exit, because the sizes of the fields in a record may vary. By using a BMP, which uses misprediction history and distance, we are able to capture the size of the fields and apply it across records of the same structure. As a result, the BMP significantly reduces the number of branch mispredictions.

4. Performance and Energy-Efficiency of Complementary Branch Predictors

In this section, we present four sets of results: 1) The reduction in the branch misprediction rate due to using a complementary branch predictor in conjunction with static, bimodal, and gshare branch predictors (Section 4.2), 2) The comparison between a loop predictor (LP) and BMP (Section 4.3), 3) The improvement in the processor performance (CPI) after adding a complementary branch predictor (Section 4.4), and 4) The energy-efficiency (EDP) of complementary branch predictors (Section 4.5).

4.1. Evaluation Methodology

To collect the results presented in this paper, we implemented the BMP in `sim-bpred` from the Simplescalar tool suite [3], version 3.0d and in the `wattch` [2], version 1.02. We used the former simulator due to its simulation speed to quantify the branch prediction accuracy of the measured branch predictors with and without the BMP, while we used the latter simulator to measure the impact on IPC and energy-delay product (EDP). To evaluate the efficacy of the BMP over the range of branch predictors that are typically found typical current-generation embedded processors and likely next-generation processors, we evaluated the following predictors: static (Predict taken for backwards branches and not-taken for forward branches), bimodal, and gshare [14]. For the bimodal and gshare branch predictors, we varied the size of these predictors to use a hardware budget of 0.25KB (1024 entries) to 4KB (16,384 entries). We chose these branch predictors and sizes based on the sizes of branch predictors in current-generation embedded processors such as Freescale Semiconductor's e300, e500, and e600 [7] processors; ARM's ARM11 and ARM12 [1] processors; MIPS's 4K, 5K, 24K, 34K, 20K processors [13], and PA Semi's PA64T [25]. Table 1 shows the processor configuration, which was heavily based on the PA64T since it is the most advanced processor, for the IPC and EDP results that are presented in this section. The size of the BMP ranged from less than 0.03KB (16 entries) to less than 2KB (1024 entries).

In this paper, we used benchmarks from the SPEC 2000 [10] and MiBench [8] benchmark suites. For the SPEC 2000 benchmarks, we downloaded pre-compiled Alpha binaries from [23]. We present the results for all 26 SPEC 2000 benchmarks when using the `reference` input set, but, for the benchmarks with more than one input set, to reduce the simulation time, we randomly selected one input set. The input set is listed in parenthesis for the following list of benchmarks that had more than one input set (input set used): *gzip* (*graphic*), *vpr* (*route*), *gcc* (*166*), *art* (*110*), *eon* (*cook*), *vortex* (*ref1*), and *bzip2* (*graphic*). For the MiBench benchmarks, due to compilation problems, we used a subset of the benchmarks (*basicmath*, *bitcount*, *dijkstra*, *fft*, *ghostscript*, *jpeg*, *patricia*, *qsort*, *rsynth*, *sha*, *stringsearch*, and *susan*). These benchmarks were compiled using a Compaq/DEC C compiler with full optimization. For all MiBench benchmarks, we used the large input set. Overall, we evaluated a total of 38 benchmarks.

Table 1. Processor Configuration

Parameter	Configuration
Issue Policy/Width	**Policy**: Out-of-order; **Width**: 4-way fetch, decode, issue, commit
Instruction Window	**Queue entries:** 16 instruction fetch queue, 64 reorder buffer, 32 load-store queue
Branch Predictor	**Misprediction latency**: 20 cycles, **Link stack entries**: 16, **Branch Target Buffer**: 512-entry, 4-way associative
Execution Units	1 branch unit, 1 load-store unit, 3/1 simple/complex integer units, 1/1 simple/complex floating-point units
MSS	**Ports**: 2, **Memory latency**: 100 cycles, **Width**: 16 bytes
Caches	**L1**: Split; 32KB, 4-way associative, 32B lines, LRU, 2 cycle latency **L2**: Unified; 1024KB, 8-way associative, 64B lines, LRU, 12 cycle latency
MMUs	**Page size**: 4KB; TLBs: Split; 128-entry, fully associative, 12 cycle latency

To reduce the simulation time of the SPEC 2000 benchmarks, we used multiple 100M instruction simulation points [22] that we generated using

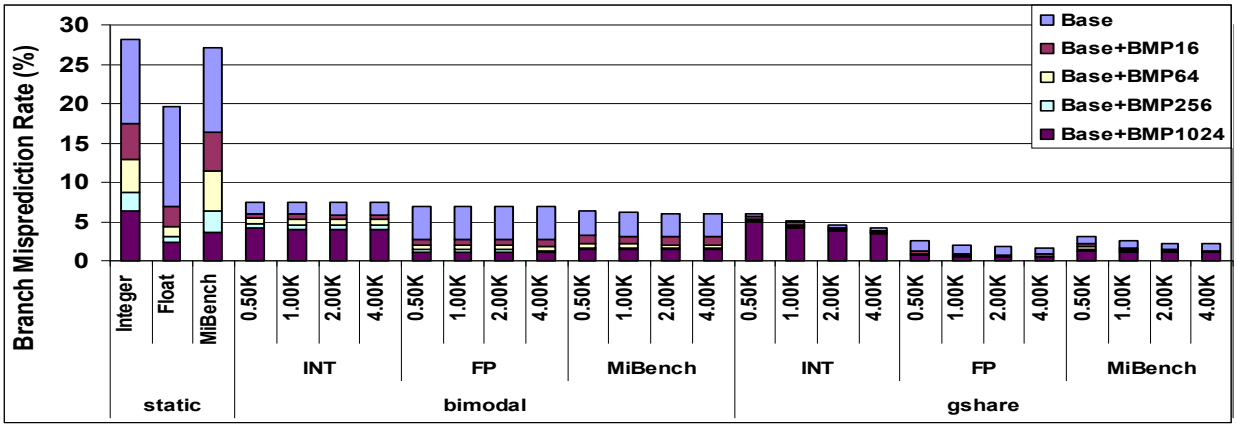

Figure 6. Reduction in the branch misprediction rate due to adding a BMP. The height of each bar represents the average branch misprediction rate without a BMP while the height of each segment shows the number of additional branch mispredictions that were removed by adding BMP entries. In this figure, smaller is better.

SimPoint 1.1 with a max_K of 10 and with 7 random seeds. We used the 100M instructions preceding the simulation interval to warm-up the branch predictor, BMP, and pipeline. Due to their relatively short simulation time, we ran the MiBench benchmarks to completion.

4.2. Reduction in the Branch Misprediction Rate Due to Adding a BMP

Figure 6 shows average branch misprediction rate for the integer, floating-point, and MiBench suites for the three branch predictors and across all hardware budgets. The height of each bar represents the average branch misprediction rate for the "Base" configuration, *e.g.*, no BMP, while the height of each segment shows the number of additional branch mispredictions that were removed by adding extra BMP entries. Therefore, the height of the bottom two segments shows the average branch misprediction rate for the Base+BMP256 configuration, while the Base+BMP256 segment shows the percentage of branches that were previously mispredicted and that are now converted to correct predictions when using a 256-entry BMP. For example, for the static branch predictor, the average branch misprediction rate for the integer, floating-point, and MiBench benchmarks is 28.2%, 19.7%, and 27.1%, respectively. Adding a 16-entry BMP reduces the average branch misprediction rates to 17.4%, 6.9%, and 16.5%, respectively (*i.e.*, the height of the bottom four segments) while adding another 48 MPBT entries, for a total of 64, further reduces the average branch misprediction rates to 12.8%, 4.3%, 11.4%, respectively (*i.e.*, the height of the bottom 3 segments).

The results in Figure 6 show several key results. First, and most importantly, complementary branch predictors can significantly reduce the branch misprediction rate for all branch predictors across a wide range of applications. The results also show that even small BMPs can significantly reduce the number of branch mispredictions, although large BMPs yield more significant reductions, albeit with diminishing returns. The results are particularly dramatic for the static branch predictor. The results show that adding a 1024-entry BMP reduces the average branch misprediction rate from 28.2%, 19.7%, and 27.1% for the integer, floating-point, and MiBench suites, respectively, to 6.4%, 2.3% and 3.6%, respectively. This result is extremely significant since many embedded processors (*e.g.*, Freescale Semiconductor's e300 and MIPS' 4K and 5K processors) use static branch prediction; instead of replacing the static branch predictor with a dynamic branch predictor (which consumes a significant amount of chip area and power and is non-trivial to integrate into the pipeline), adding a complementary branch predictor like the BMP can yield branch prediction accuracy that is similar to that of dynamic predictors, but with significantly less design effort.

Although increasing the hardware budget of the branch predictor reduces the branch misprediction rate, the results in Figure 6 show that allocating part of that bit budget to a BMP instead will, in most cases, reduce the branch misprediction rate by a larger amount than devoting the entire bit budget to a larger conventional branch predictor. For example, for the MiBench benchmarks, increasing the size of the branch predictor from 0.25KB to 4KB reduces the misprediction rate from 3.9% to 2.1%. By contrast, adding a 256-entry MPBT to the 0.25KB configuration (for a total bit

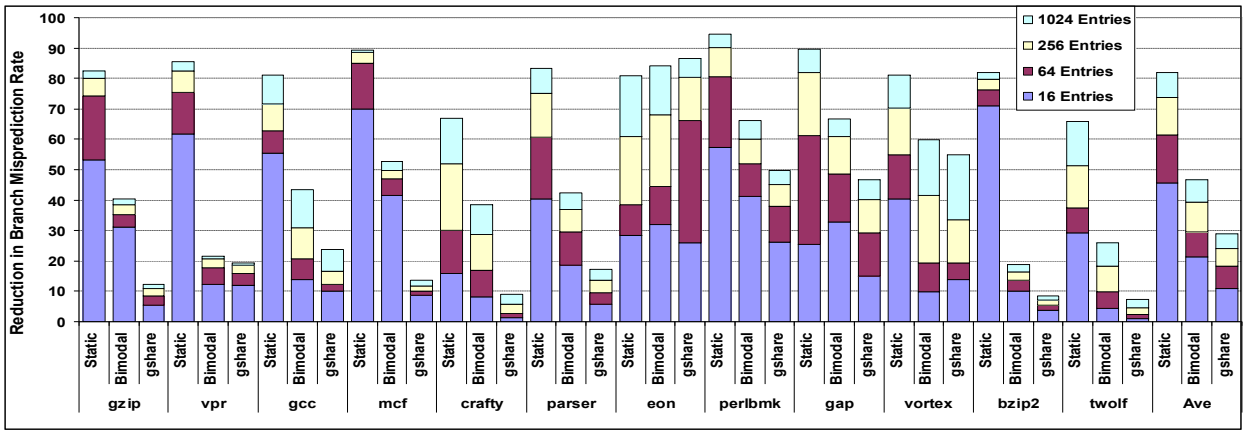

Figure 7. Percentage reduction in branch misprediction rate with BMP (baseline bimodal or gshare branch predictor is 1KB) for SPECint 2000 benchmarks.

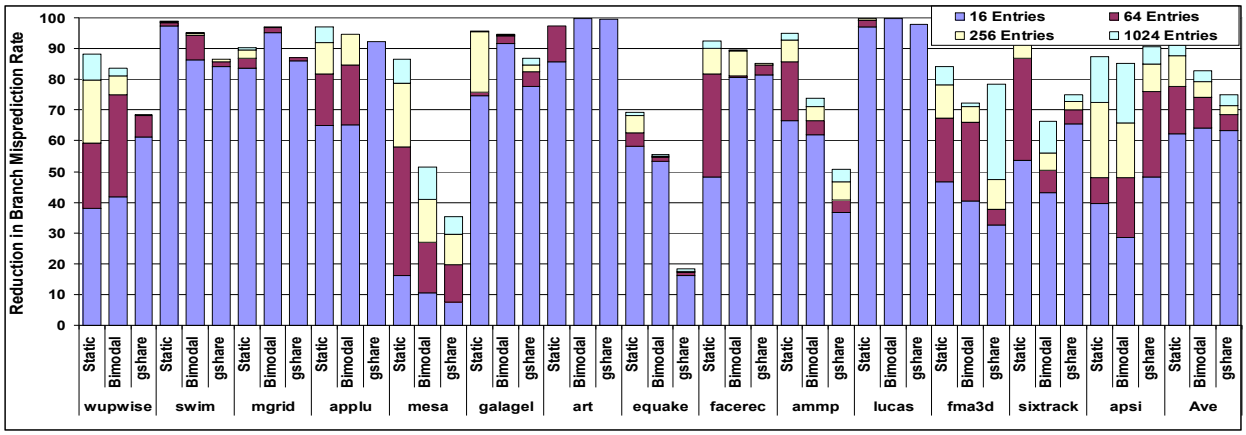

Figure 8. Percentage reduction in branch misprediction rate with BMP (baseline bimodal or gshare branch predictor is 1KB) for SPECfp 2000 benchmarks.

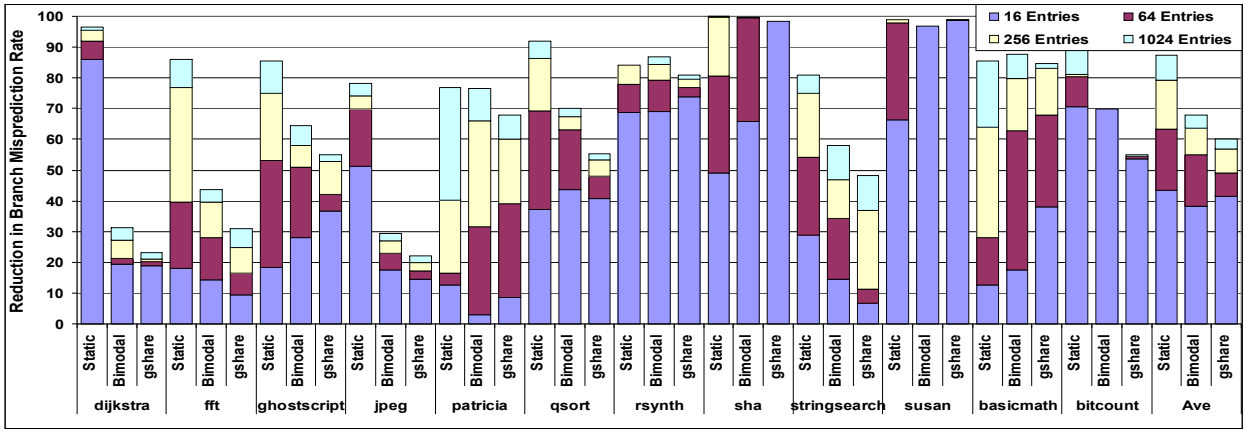

Figure 9. Percentage reduction in branch misprediction rate with BMP (baseline bimodal or gshare branch predictor is 1KB) for MiBench benchmarks.

budget of 0.69KB (0.25KB for the branch predictor and 0.44KB for the BMP)) reduces the average branch misprediction rate to 1.7%. Therefore, as this result shows, adding a small BMP to a conventional branch predictor can simultaneously reduce both the total bit budget and the branch misprediction rate. Furthermore,

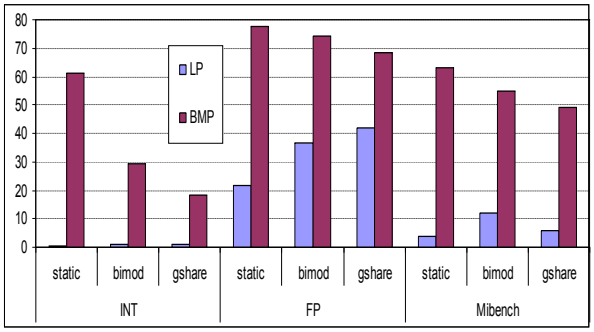

Figure 10. Percentage reduction in the branch misprediction rate with a 128 byte loop predictor (LP) or BMP. The sizes of the baseline dynamic branch predictors are 1KB.

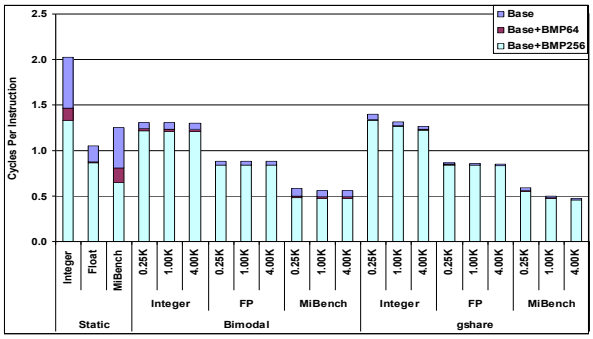

Figure 11. Reduction in CPI due to adding a BMP

since the BMP is not on the processor's critical path (as is the case for a conventional branch predictor) and since MPBT is not accessed every cycle, BMP-based branch predictors should also have lower branch predictor access latencies and lower power consumption.

Finally, the results in Figure 6 show that although the BMP can reduce the branch misprediction rate for all suites, it yields the largest benefit for the floating-point benchmarks, followed by the MiBench and integer suites. Figures 7, 8, and 9 show the reduction in the branch misprediction rate due to the BMP for individual benchmarks.

4.3. Comparison with Loop Predictor

Before presenting more results, it is important to note that BMP does not only target loop branches, which is the case for loop predictors [9] (LP). There are several significant differences between the BMP and a LP. First of all, the BMP is not limited only to loops, but rather can target all types of branches. Second, the LP uses local history to make predictions while the BMP uses different types of global history (*e.g.*, global misprediction history) to make its predictions. Third, the LP makes a prediction for every branch, while the BMP only makes a prediction after a mispredicted branch. Therefore, the BMP is accessed less frequently, which is important for power reasons, and is not on the processor's critical path. Finally, since the LP competes with other constituent predictors within the branch predictor, even if it is chosen as the highest confidence prediction, its prediction may not be different than the predictions from the other constituent predictors, *i.e.*, may predict correctly anyways. By contrast, the BMP only makes predictions for branches that are frequently mispredicted, *i.e.*, unlikely to be predicted correctly.

To quantify the performance difference between a LP and a BMP, we implemented the loop predictor that was described in [21]. Figure 10 shows the percentage reduction of the branch misprediction rates when a 128 byte LP or BMP added to a static, a 1KB bimodal, or a 1KB gshare branch predictor. The results show that the BMP reduces the branch misprediction rate more significantly than the LP does. The difference is most pronounced for SPECint, where loop predictor performs poorly.

4.4. Reduction in the CPI Due to Adding a BMP

Figure 11 shows CPI for the processor configuration given in Table 1 for the base branch predictor and with a 16-entry and 256-entry BMP. The format for Figure 11 is similar to that of Figure 6 in that the height of the bar represents the CPI of the base processor configuration while the heights of the bottom-most and second-from-the-bottom segments represent the CPI after adding 256-entry and 16-entry BMPs, respectively. Due to time constraints, in this sub-section and the next two, we only present the results for 3 branch predictor sizes (0.25KB, 1KB, and 4KB) and for 2 BMP sizes (64 and 256 entries).

The results in Figure 11 show that adding a 64-entry BMP to the static predictor yields average speedups of 20.0% to 53.6%, while a 256-entry BMP yields average speedups of 21.7% to 93.0%. These results indicate that complementary branch predictors like the BMP yield significant performance improvements, which coupled with its small size (112 bytes for the 64-entry BMP) and its ease of implementation into the pipeline (off of the critical path and accessed only after mispredictions), makes it a very attractive add-on for embedded processors with static branch predictors.

For the 1KB bimodal branch predictor, the average speedup due to a BMP of 64 (256) entries ranges from 4.5% to 14.6% (5.1% to 18.3%), while the corresponding average speedups for the 1KB gshare branch predictor range from 2.0% to 3.4% (2.1% to

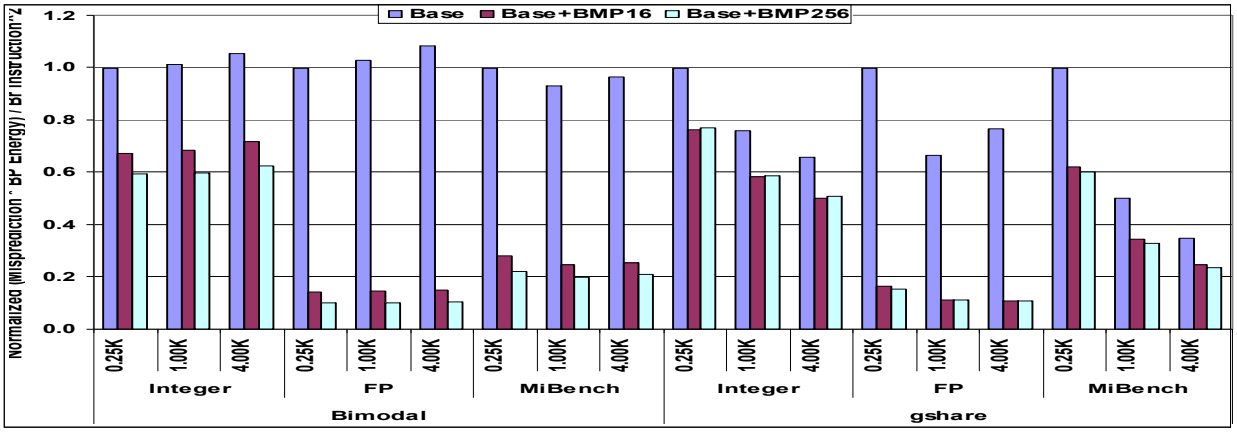

Figure 12. Improvement in branch predictor energy-efficiency due to adding a BMP. In the figure, lower is better.

5.33%). Although the average speedups when the gshare branch predictor is the baseline predictor are not as impressive as the speedups for static and bimodal, the speedup is greater than 10% for several benchmarks (*e.g.*, *gzip*, *eon*, and *bitcount*) when adding a 256-entry BMP.

The key results from Figure 11 are: 1) The BMP improves the performance for all three branch predictors and for all three branch predictor sizes, 2) Larger BMPs yield larger performance improvements than smaller ones, but with diminishing returns, and 3) For some branch predictor configurations, using part of the bit budget for a BMP yields higher performance than devoting the entire bit budget to a conventional branch predictor, *e.g.*, 1) 0.25KB bimodal with a 64-entry BMP vs. a 1KB or 4KB bimodal or 2) 1KB gshare with a 256-entry BMP vs. 4KB gshare for the integer and floating-point benchmarks.

4.5. Improvement in the Energy-Efficiency Due to Adding a BMP

The results in Sections 4.2 and 4.4 present the performance potential of the BMP, in terms of reducing the branch misprediction rate and the execution time (CPI). Recall from Section 2 that, in contrast with conventional branch predictors, the BMP is accessed only after branch mispredictions, or, in other words, relatively infrequently. Fewer accesses mean that the BMP is more energy-efficient. Note that when the branch predictor has a higher branch prediction accuracy, the number of BMP accesses drops proportionally. In essence, there is an implicit control mechanism built into the BMP to reduce its energy consumption when it is not needed to improve the branch prediction accuracy.

The remainder of this section presents the potential for energy reduction by using a BMP. All of the results that we present in this section were based on using `cc3` clocking in `wattch`.

4.5.1. Energy-Efficiency of Branch Predictors with Complementary Branch Predictors

Figure 12 shows the improvement in the branch predictor's energy-efficiency due to adding the BMP. To quantify the branch predictor's energy-efficiency, we use the product of the branch misprediction rate and the branch predictor's energy per (committed) branch instruction, or:

$$\text{Branch Predictor Energy Efficiency} = \frac{\text{Mispredicted Branches}}{\text{Committed Branches}} \times \frac{\text{Branch Predictor Energy Consumption}}{\text{Committed Branches}}$$

Note that, when using a BMP, the energy consumption of the branch predictor includes both the power of the branch predictor and the BMP. To make the comparison of energy-efficiency easier across the different branch predictor sizes, for each branch predictor, we normalize the energy-efficiency with respect to the energy-efficiency for the Base 0.25KB branch predictor. Therefore, for each branch predictor and suite, the Base 0.25KB bar will always be 1. Consequently, in this figure, since the static branch predictor essentially consumes zero energy, normalization is impossible, so we omit the presentation of the static branch predictor configurations.

The results in Figure 12 yield several key results. First, since the BMP is significantly smaller and less frequently accessed than the branch predictor and since the results in Section 4.2 show that it significantly reduces the branch misprediction rate, it is not surprisingly to see that BMP-based branch predictors are significantly more energy-efficient than the same

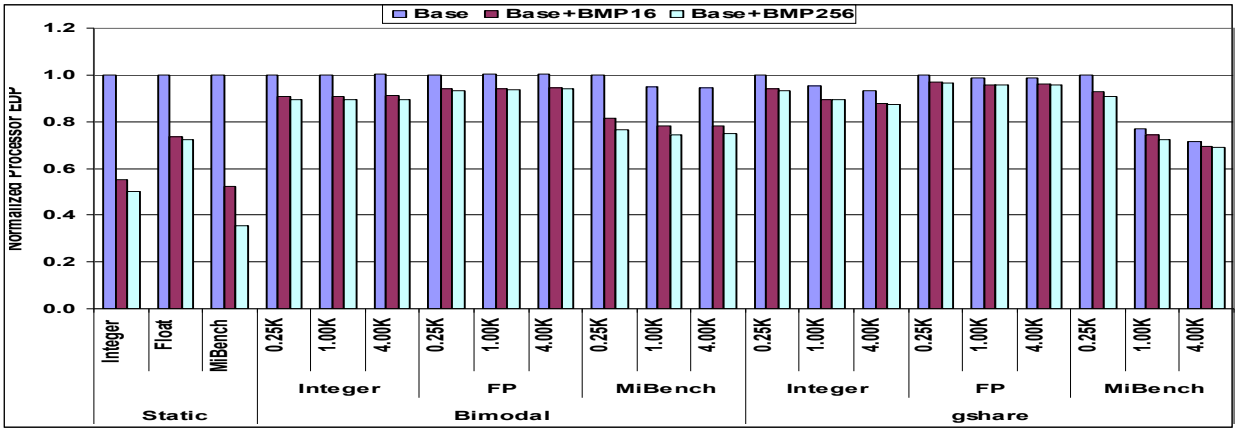

Figure 13. Improvement in processor energy-efficiency due to adding a BMP

size branch predictor without a BMP, even though BMP-based branch predictors are slightly larger. More specifically, a 0.25KB bimodal branch predictor with a 256-entry BMP is 40.6%, 89.8%, and 78.3% more energy-efficient than a 0.25KB bimodal branch predictor without the BMP for the integer, floating-point, and MiBench benchmarks, respectively. For a 4KB gshare branch predictor, adding a 256-entry BMP yields a 22.9% 85.9%, and 32.3% improvement in the energy-efficiency of a 4KB gshare branch predictor without a BMP.

Second, in most cases, a smaller branch predictor with a BMP is more energy-efficient than a larger branch predictor without a BMP. For example, for the MiBench benchmarks, a 1KB gshare branch predictor with a 256-entry BMP is 6% more energy-efficient than a 4KB branch predictor without the BMP. Furthermore, across all the benchmarks, a 0.25KB bimodal branch predictor with 64-entry BMP is substantially less power-hungry than a 4KB bimodal branch predictor without BMP.

Third, with only two exceptions (0.25KB and 4KB gshare branch predictor for the integer benchmarks), increasing the number of BMP entries from 64 to 256 improves the overall energy-efficiency of the branch predictor. By contrast, without a BMP, increasing the bit budget of the branch predictor only improves its energy-efficiency in half of the cases. The key conclusion from this result is that not only are complementary branch predictors more scalable from a timing point-of-view (since they are not on the critical path), but they are more scalable from an energy-efficiency point-of-view.

4.5.2. Energy-Efficiency of Processors with Complementary Branch Predictors

Figure 13 shows the improvement in an embedded processor's energy-efficiency when using a BMP as compared to not using a BMP. The results in Figure 13 are normalized in the same way as were the results in Figure 12, namely, with respect to the 0.25KB branch predictor for each configuration and suite. In this figure, the metric for energy-efficiency is energy-delay product (EDP).

From Figure 13, we see that processors with a BMP are more energy-efficient than processors without a BMP, for the same size branch predictor. For processors with a static predictor, adding a 256-entry BMP improves the processor's energy-efficiency by 49.6%, 27.7%, and 64.6% for the integer, floating-point, and MiBench benchmarks. For a 1KB bit budget, the energy-efficiency of the processor for the other two branch predictors for 256-entry BMP, ranges from 2.8% (gshare, floating-point) to 20.3% (bimodal, MiBench).

Furthermore, with the exception of the gshare branch predictor and the MiBench benchmarks, processors that use a 256-entry BMP are 1.8% to 21.0% energy-efficient than processors without a BMP, but that have a branch predictor that is four times larger. For the MiBench benchmarks, the energy-efficiency of a 0.25KB and 1KB gshare branch predictor with a 256-entry BMP is 17.4% and 0.9% lower than a 1KB and 4KB, respectively, gshare branch predictor. However, it is important to note that the latter processors with have larger chip areas (and consequently higher fabrication costs) and possibly higher branch predictor access times (which are not modeled here).

5. Related Work

Sendag *et al*. [19] introduced the concept of complementary branch predictors and developed a mechanism that helped to correct future branch mispredictions without increasing the size of the branch predictor. This paper builds and extends on that

idea in several ways. First, it studies the predictability of branch mispredictions. Second, it analyzes the potential of using complementary branch predictors with branch predictors that are typically found in embedded processors. Third, it quantifies the performance impact (CPI) that a BMP can have on the processor. Fourth, it evaluates the potential speedup in the execution time due to using a complementary branch predictor in an embedded processor. Fifth, it analyzes the power efficiency of BMP-based branch predictors. Finally, it analyzes the code to determine why the BMP can successfully predict frequently mispredicted branches.

The majority of power reduction techniques concentrate on eliminating dynamic power requirement of branch predictor and BTB. Parikh et al. [16] propose eliminating unnecessary lookups to the branch predictor and BTB for non-branch instructions using hardware filters or compiler-generated hints. It also suggests that exploiting banked branch predictor organizations can reduce branch predictor access time and power. In [18], they propose a software-programmable BTB, named ACBTB, which can adapt to application control flow information extracted by the compiler/linker. They claim that this mechanism can also guarantee an access hit in branch target resolution. Baniasaidi and Moshovos propose using a branch prediction predictor [4] and a selective predictor access (SEPAS) [5], in addition to gating the access of branch predictor and BTB, to eliminate accesses to the sub-predictors in a hybrid branch predictor by exploiting the temporal and sub-predictor locality of branches. SEPAS also removes branch predictor and BTB updates that store information that already exists in the predictor, further lowering the power requirement of branch predictor and BTB. Chaver *et al.* in [6] propose a similar scheme which gates the access to hybrid branch predictor and uses an adaptive BTB which dynamically downsizes itself for applications that are not BTB-size sensitive.

In addition to reducing the dynamic power dissipation, another important avenue of research focuses on reducing the static power leakage. Hu *et al.* [11] deactivate the branch predictor, sub-predictor and/or branch predictor banks when there are no branch instructions to be predicted. While this mechanism reduces power consumption, access to a hibernating predictor block results in extra wake-up delay. To mitigate this problem, Monchiero *et al.* propose to introduce new hint instruction, HI, for upcoming branches [15]. The decoder logic activates the branch predictor as soon as a HI instruction is fetched. This instruction also triggers the branch predictor to use the information encoded in the instruction to determine if the target is readily available. Yang and Orailoglu [24] use compiler hints to pre-activate the branch predictor and BTB.

The aforementioned techniques focus on directly lowering the power consumption of branch predictors, dynamically or statically. In [17], Pasricha and Veidenbaum analyze this problem from the application/system operation point-of-view. They examine the effects of context switch on the branch predictor accuracy and show that context switch introduces branch predictor and BTB pollution. Based on this observation, they propose a storage-efficient mechanism to dump branch predictor's contents during a context switch to reduce pollution.

In stark contrast to all of the previous work, our BMP design attacks the power efficiency problem from a different angle. We combine a smaller branch predictor, simple or complex, with a simple BMP to achieve high (or higher) prediction accuracy. Since this approach is orthogonal to the aforementioned techniques, it can be used in conjunction with them. Our results show that our approach not only can improve the energy-efficiency of the branch predictor/processor like the other techniques, but can also improve the branch prediction accuracy/IPC performance, which is separates it from the other techniques.

6. Conclusion

Since embedded processors need to have branch predictors with high branch prediction accuracy to achieve good performance, computer architects have proposed branch predictors that are increasingly more complex and/or larger. Unfortunately, the cost of these more aggressive branch predictors are higher prediction latencies and misprediction penalties, which offset the higher prediction accuracy, in addition to larger chip area and increased power consumption.

To address this problem, in this paper, we propose adding complementary branch predictors to the existing branch predictors of embedded processors. By concentrating on making predictions for only the subset of branches that degrade the processor's performance, a complementary branch predictor requires less area and is accessed less frequently. Consequently, it is inherently more power efficient.

To quantify the efficacy of this approach, we implemented a branch misprediction predictor (BMP), which uses the branch misprediction history to predict which future branch will be mispredicted next and when that will occur. The BMP then flips the branch predictor's predicted direction for this branch. Since we do not alter branch predictor's working mechanism, a complementary branch predictor can improve the accuracy of any branch predictor. Moreover, it can also

be used in conjunction with other energy reduction techniques for branch predictors, such as access gating for branch predictors and BTB as well as dynamically deactivating the branch predictor and its sub-components.

Our results show that adding a small 16-entry (28 byte) BMP reduces the branch misprediction rate of static, bimodal, and gshare branch predictors by an average of 51.0%, 42.5%, and 39.8%, respectively, across 38 SPEC 2000 and MiBench benchmarks. Furthermore, a 256-entry BMP yields an average speedup up to 67.3% and improves the energy-efficiency of the branch predictor and processor up to 97.8% and 23.6%, respectively.

Acknowledgments

This research is supported in part by US National Science Foundation grant CCF-0541162, the University of Minnesota Supercomputing Institute.

References

[1] http://www.arm.com

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *International Symposium on Computer Architecture*, 2000.

[3] D. Burger and T. Austin, "The Simplescalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*, 1997.

[4] A. Baniasadi and A. Moshovos, "Branch predictor prediction: a power-aware branch predictor for high performance processors," *International Conference on Computer Design*, 2002.

[5] A. Baniasadi and A. Moshovos, "SEPAS: A highly accurate energy-efficient branch predictor," *International Symposium on Low Power Electronics and Design*, 2004.

[6] D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang, "Branch prediction on demand: an energy-efficient solution," I*nternational Symposium on Low Power Electronics and Design*, 2003.

[7] http://www.freescale.com

[8] M. Gauthus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *Workshop on Workload Characterization*, 2001.

[9] Gochman *et al*. "The Intel Pentium-M processor: Microarchitecture and performance," *Intel Technology Journal*, 7(2), pp. 21-33, 2003.

[10] J. Henning, "SPEC CPU CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, Vol. 33, No. 7, July 2000, pp. 28-35.

[11] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi, "Applying decay strategies to branch predictors for leakage energy savings," *International Conference on Computer Design*, 2002.

[12] D. Jiménez, "Piecewise Linear branch prediction," *International Symposium on Computer Architecture*, 2005.

[13] http://www.mips.com

[14] S. McFarling, "Combining Branch Predictors," *Digital Western Research Laboratory Technical Report TN-36M*, 1993.

[15] 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," *The VLSI Journal*, Vol. 38, No. 3, pp. 515-524, Jan. 2005.

[16] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, "Power issues related to branch prediction," *International Symposium on High-Performance Computer Architecture*, 2002.

[17] S. Pasricha and A. Veidenbaum, "Improving Branch Prediction Accuracy in Embedded Processors in the Presence of Context Switches," *International Conference on Computer Design*, 2003.

[18] P. Petrov and A. Orailoglu, "Low-power branch target buffer for application-specific embedded processors," *IEE Transactions on Computers and Digital Techniques*, Vol. 152, No. 4, pp. 482-488, July 2005.

[19] R. Sendag, J. Yi, and P. Chuang, "Branch Misprediction Prediction: Complementary Branch Predictors," *IEEE Computer Architecture Letters*, 2007.

[20] A. Seznec, "Analysis of the O-GEHL predictor," *International Symposium on Computer Architecture*, 2005.

[21] A. Seznec, "A 256 Kbits L-TAGE branch predictor," *Journal of Instruction Level Parallelism*, 2006.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[23] http://www.23.com/benchmarks.html

[24] C. Yang and A. Orailoglu, "Power Efficient Branch Prediction through Early Identification of Branch Addresses," *International conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[25] T. Yeh, "The Low-Power High-Performance Architecture of the PWRficient Processor Family," *Hot Chips*, 2006.