Energy Efficient Branch Prediction on the Cell SPE

Martijn Briejer, Cor Meenderinck, and Ben Juurlink
Computer Engineering Lab
Faculty of Electrical Engineering Mathematics and Computer Science
Delft University of Technology, the Netherlands
{*mbriejer, cor, benj*}*@ce.et.tudelft.nl*

***Abstract*—We propose novel power efficient branch predictors for the Cell SPU, which normally depends on compiler inserted hint instructions to predict taken branches. Several prediction schemes were designed all using a Branch Target Buffer (BTB) to store the branch target address and the prediction, which is computed using a bimodal counter. One prediction scheme pre-decodes instructions once they are fetched from the local store and accesses the BTB only for a branch instruction. Several ways to combine the previous with the existing hint instructions are studied. We also introduce branch warning instructions which initiates a branch prediction before the branch instruction is even fetched. It allows to fetch the instructions starting at the branch target address and completely removes the branch penalty for correctly predicted branches. Assuming a 256 entry BTB a speedup of up to 18.8% was achieved. The power consumption of the branch prediction schemes is estimated to be 1% or less and the average energy delay product is reduced by up to 6.2%.**

***Keywords*-branch prediction, cell spu, energy efficient**

I. Introduction

Recently there is a clear trend towards multi-core processors. To meet the power constraints, each core has to be very energy efficient. One of the most advanced multi-core processor is the Cell broadband engine. Because of its energy and area efficient design, it contains one general purpose core, the Power Processing Element (PPE), and eight Synergistic Processing Elements (SPEs).

To further improve the Cell's performance, researchers have proposed various modifications to the SPEs. Different parts of the SPE were targeted, from the memory system [1] to the instruction set [2]. However, no research was done to investigate the possible performance increase of hardware branch prediction.

In this work, we propose three branch predictors. Energy efficiency is obtained by reducing the number of predictions. The Simple Bimodal Predictor (SBP) does a prediction for branch instructions only. Normally, a branch predictor does a prediction for every instruction that enters the pipeline. The SBP identifies branch instructions early in the pipeline by pre-decoding instructions. We created a second version of this predictor that also uses hint instructions. The third branch predictor uses branch warning instructions to identify branches, in combination with hints. As a reference to the maximum obtainable speedup by branch prediction, we implemented an aggressive branch predictor, that does a prediction for each instruction directly when it is fetched from the local store.

Several techniques have been proposed before to reduce the energy consumption of branch predictors, which can be up to 10% for modern processors [3]. Banking the branch predictor table reduces the active part when doing a lookup, thus reducing the energy consumption. Parikh et al. [3] searched for an optimal banking strategy. They also propose a Predictor Probe Detector (PPD), which pre-decodes instructions in the instruction cache to detect a branch. A Branch Target Buffer (BTB) lookup is done only for a branch, which reduces the energy consumption by approximately 45%. Kahn and Weiss [4] reduced the number of BTB lookups by using a counting Bloom filter, that determines if an address is in the BTB or not. To minimize the extra delay, a small direct mapped BTB is used in parallel. Together those provide a 51% reduction of the dynamic power consumption.

Yang and Orailoglu [5] proposed a Branch Identification Unit (BIU), which controls access to the BTB. It uses statically extracted program control flow information inserted by the compiler to predict branches early. Chaver et al. [6] used profile information to adapt the predictor hardware on the fly, using configuration instructions added by the compiler. Monchiero et al. [7] proposed to use compiler inserted hint instructions to inform the VLIW-processor that a branch is coming. If the target is known, the branch can be taken without penalty.

The techniques described above are designed for several architectures, but not for the Cell SPE that we target. We used some ideas and implemented them with some modifications in the SPE. First, our simple bimodal predictor pre-decodes branch instructions like the PPD, but we do not use a separate table to store the result. Instead we incorporated this in the pipeline. We also combined the predictor with hint instructions, which has not been done before. Second, we use the idea of inserting hint instructions for branches with unknown target in our branch warning predictor. However, in our case the predictor can overrule hints if it predicts 'not taken'.

The remainder of this paper is organized as follows. Section II provides an overview of the SPE architecture. Sec-

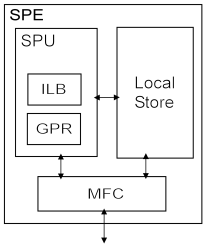

Figure 1. Overview of the Cell SPE.

tion III describes the experimental setup. In Section IV we describe the proposed branch predictors. The performance and energy efficiency results are presented in Section V. Finally, Section VI concludes the paper.

II. The SPU Architecture

The Cell processor consists of one PPE (Power Processing Element) and eight SPEs (Synergistic Processing Elements) [8]. The PPE is a general purpose PowerPC that runs the operating system and controls the SPEs. The SPEs function as accelerators; they operate autonomously but are depending on the PPE for receiving threads to execute. The SPE consists of a Synergistic Processing Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC) as depicted in Figure 1. The SPU has direct access to the LS, but global memory can only be accessed through the MFC by DMA commands. As the MFC is autonomous, a double buffering strategy can be used to hide the latency of global memory access. While the SPU is processing a task, the MFC is loading the data needed for the next task into the LS.

The SPU has 128 registers, each 128 bits wide. All data transfers between the SPU and the LS are 128-bit wide. Also the LS accesses are 128-bit aligned. The ISA of the SPU is completely SIMD and the 128-bit vectors can be treated as one quadword (128-bit), two doublewords (64-bit), four words (32-bit), eight halfwords (16-bit), or 16 bytes.

Instructions are fetched from the LS into the Instruction Line Buffer (ILB) in groups of 32, fitting in one line. The ILB can store up to 3.5 lines. The SPU has six functional units, each assigned to either the even or odd pipeline. The SPU can issue two instructions concurrently if they are located in the same doubleword and if they execute in a different pipeline. Instructions within the same pipeline retire in order.

The SPU has no dynamic hardware branch predictor. In case of branches, the SPU continues execution sequentially. A branch miss has a penalty of 18 cycles. The compiler can insert hint instructions to predict a branch taken, if it can determine the branch target at compile time. If the hint is executed 16 cycles before the branch, execution can continue without delay if the hint is correct.

Despite the usage of hint instructions, many programs have a significant number of branch miss stall cycles. There are several reasons for that. First, not every branch that should be hinted, can be hinted; only one hint can be active at a given time, thus if two branches are to close only one of them is hinted. Second, hints are static and cannot change during the executing of a program. Thus if a hint is wrong, it stays wrong. Finally, branches that are not hinted can be taken too. To improve the branching capabilities of the SPU we propose to extend it with a power efficient hardware branch predictor.

III. Experimental Setup

We used the CellSim [9] simulator to implement and test our branch predictors. CellSim is a modular simulator build in the Unisim environment and is very suitable for architectural research. Before extending CellSim, we first added support for hint instructions, which was not present yet.

CellSim is an instruction set simulator, and therefore not all parts of the SPU are modelled cycle accurate. However, lots of configuration parameters can be changed in order to get the performance close to reality. For all benchmarks used throughout this work, we optimized the configuration. The performance was validated using performance statistics from another simulator, IBM SystemSim, which is a cycle accurate simulator. The results of this performance validation are depicted in Table I. It shows that in all cases the error is less than 5%.

Table I
Validation of CellSim against IBM SystemSim. Execution in cycles and the relative error are stated.

Benchmark	SystemSim	CellSim	Error
MiniGZip	22431156	2300309	2.5%
Listrank	18437301	18288224	-0.8%
ListrankP3	15085386	15662336	3.8%
MergeSort	672244	685548	2.0%
MergeSort Rnd	712395	744114	4.5%
QuickSort	363944	377308	3.7%
QuickSort Rnd	538141	551716	2.5%
SPE-JPEG	3278101	3132410	-4.4%
DB Filter	2372687	2363310	0.3%

We selected benchmarks from different computing area's. Because CellSim does not run an operating system and not all system calls are implemented, the available benchmarks were limited. Also, the simulation of the benchmarks should finish in reasonable time and the performance statistics should have a significant number of branch miss stall cycles in order to have some room for improvement. In our opinion, the selected benchmarks are a good representation of applications suitable for the Cell processor.

IV. Proposed Predictors

In this section we describe the different branch predictor implementations. They are all based on a Bimodal Branch

Predictor (BBP). The latter uses a bimodal counter to make a prediction. As depicted in Figure 2, the prediction consists of two bits. The first bit determines if a branch is taken or not taken, while the second bit indicates if the prediction is strong or weak. When a branch instruction is executed, the prediction is updated by adding one if it was taken or subtracting one if not. By default, the prediction is weakly not taken.

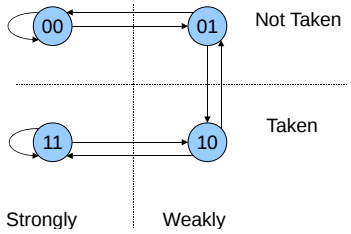

Figure 2. State Diagram of the Bimodal Counter.

The prediction is stored in a Branch Target Buffer (BTB) as depicted in Figure 3. The BTB is indexed by the least significant bits of the branch instruction word address. The remaining bits are used as a tag to identify the branch and thereby preventing aliasing. The branch target word address is also stored.

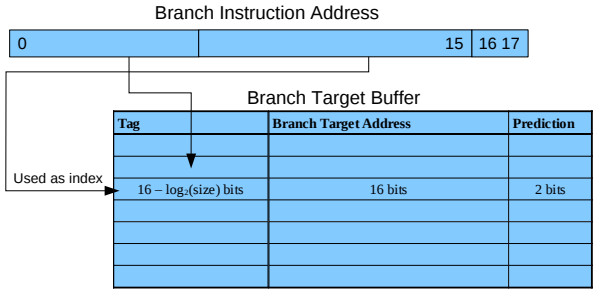

Figure 3. Design of the Branch Target Buffer.

A. Simple Bimodal Predictor

The first implementation that uses the BBP is the Simple Bimodal Predictor (SBP). For energy efficiency, a BTB lookup is done only for branch instructions. In order to do that, the SPU needs to identify branch instructions. Normally, instructions are decoded in stage 9 of the SPU pipeline (see Figure 4, thus in that stage the SPU knows the type of instruction. However, to improve performance we decided to add some hardware to the ILB, which detects a branch instruction while it is still in the ILB, by partially pre-decoding it. This is done in stage 6 of the pipeline. In the next cycle, the prediction can be done. If the branch is predicted taken, the ILB is flushed and the instructions at the branch target address are fetched from the local store. A correctly predicted taken branch now has a seven cycle penalty instead of the original 18 cycles. Hint instructions are ignored.

B. SBP combined with hints

The SBP ignores hint instruction, but they contain valuable information about the branch. If a (correct) hint is executed early enough, the branch can be taken without delay, which is better than the seven cycle penalty of the SBP. Therefore we also made an implementation of the SBP that uses hints. However, sometimes hints are incorrect. Therefore, we implemented different hint policies. Besides always using the hint (referred to as SBP-H), we also let the predictor overrule the hint if it predicts strongly not taken (referred to as SBP-OH-NLS). If a hint is overruled its target is not prefetched. As the results will prove, this policy provided the best result.

C. Branch Warning Predictor

In order to further improve the performance, we want to do the prediction earlier than the seventh cycle. Therefore we introduce a new instruction: the branch warning. This instruction functions similarly to the hint instruction, except it is inserted for a branch with uncertain target. They are inserted well in front of the branch by the compiler. If the branch warning is executed, a BTB lookup is performed. If the branch is predicted taken, the instructions at the target address are prefetched into an extra line in the ILB. Now the SPU can continue without delay, if the branch was correctly predicted, and if the branch warning was executed more than 16 cycles before the actual branch instruction. Hint instructions are also used, however they are overruled when the predictor predicts strongly not taken. Thus the predictor is accessed for a branch warning or a hint instruction, and branches that do not have either of them are not predicted. The execution of the extra branch warning instructions can cost additional cycles. However the SPU is a dual issue processor, thus the number of extra cycles could be relatively small and should be less than the cycles gained by predicting earlier. This branch prediction scheme is referred to as BWP-OH-NLS.

In our implementation, the branch warnings are implemented as hint instructions with target 0, because that required only a small amount of compiler modifications. Although in hardware treated differently, the compiler cannot distinguish them and treats both as a hint instruction. As a consequence the current branch warning insertion algorithm is not optimal.

D. Aggressive Bimodal Predictor

To investigate how much the performance potentially can be improved using a bimodal branch predictor, we also implemented a more aggressive version. The main difference with the previous predictors is that the Aggressive Bimodal

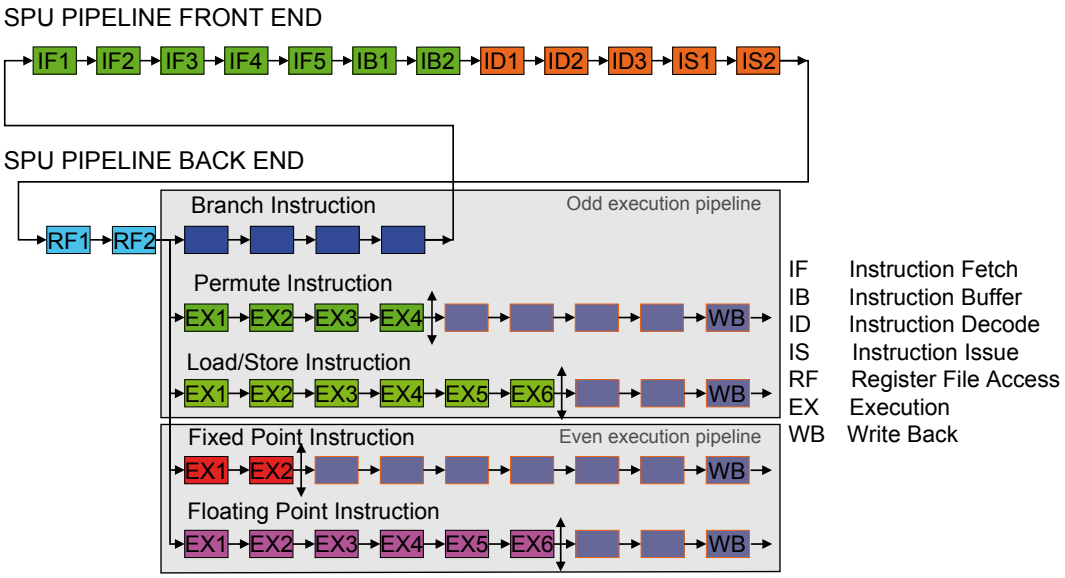

Figure 4. SPU pipeline diagram (taken from [10]).

Predictor (ABP) does a BTB lookup for every instruction. Because now there is no need to detect a branch before doing the prediction, the prediction can be done in stage 1 of the pipeline, when the instruction is fetched from the local store. Because instructions enter the execution pipeline in groups of two, two lookups are done every cycle. To prevent stalling of the pipeline because multiple successive branches are predicted taken, the ILB is extended with 8 lines, that can store the targets of 8 speculatively taken branches.

Because this implementation is much more complex than the others and uses more energy and area, it is not a realistic candidate for extending the SPU. However, it was added to this analysis as a performance reference for the other branch predictors.

V. Results and Evaluation

In this section we present the results obtained with our branch predictors. First we evaluate the performance and second we discuss the energy consumption.

A. Performance

Figure 5 shows the speedup of the different predictors compared to the original SPU, using the selected benchmarks and a 256-entry BTB. It also contains the average speedup for each predictor.

As expected, the aggressive branch predictor provides the largest speedup for all benchmarks. On average, its speedup is 15.5%. QuickSort has the largest speedup, namely 55%. There are two reasons for that. First, it has about 45% branch miss stall cycles, which is much more than the other benchmarks. Second, the code consist of small loops, which can be handled very efficiently because the ABP can have 8 outstanding branches. The other predictors cannot work that much in front, and thus they have a lower speedup. When a random input is used (QuickSort Rnd) instead of a reversed list (QuickSort), the branches are less predictable and thus the performance is lower. The same difference can be seen between MergeSort and MergeSort Rnd. The DB Filter branching behavior also depends on the input and cannot be predicted very accurate. Therefore the speedup is only 3.4%.

The SBP provides a speedup for three benchmarks only. In the original implementation, Listrank has a lot of branches that are hinted wrongly, but the SBP predicts them correctly. MergeSort has a lot of branches that are taken but not hinted, which are now correctly predicted by the SBP. The other benchmarks show no significant difference or even a slowdown. Therefore the average speedup is only 1.4%. The main reason for that is that hints are ignored. If the SBP is combined with hints (SBP-OH-NLS), the speedup is much higher, namely 7.3% on average. Using the advantages of both a predictor and hints provides good results. With this combination there is only a slowdown for MergeSort Rnd. Except for the QuickSort and MergeSort benchmarks, the performance is also close to the ABP.

The branch warning predictor (BWP-OH-NLS) shows mixed results. For Mergesort and Listrank it is faster than the SBP with hints (SBP-OH-NLS), but for MiniGZIP, SPE-JPEG, QuickSort, and DB Filter it is even slower than the SBP without hints (SBP). However, on average the branch warning predictor has a speedup of 4.7%, which is in between both SBP predictors. The mixed results are due to the non-optimal algorithm the compiler uses to

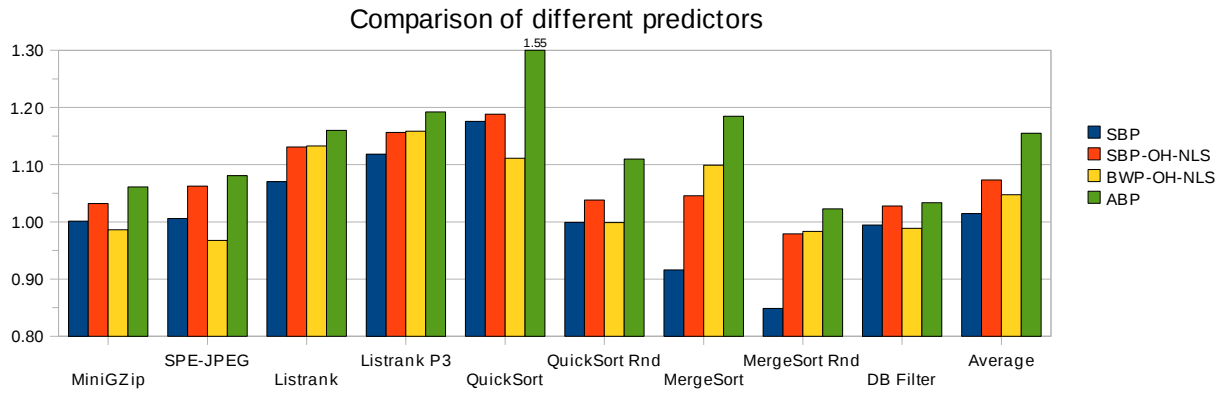

Figure 5. Comparison of the proposed branch predictors with 256 entry BTB.

insert the branch warnings. The compiler cannot distinguish branch warnings from hints, because branch warnings are implemented as hints with target 0. Because only one hint can be active at a time, the insertion of branch warnings interferes with hints. Listing 1 shows a part of the original MiniGZip code. The hint on line 1 belongs to the loop-branch on line 7, which is taken most of the time. Listing 2 shows the same code with a branch warning. Now the loop-exit-branch on line 10 has a warning, but the loop-branch has no hint anymore, which introduces a lot of extra branch miss stall cycles. Thus, to get the best performance out of this predictor, the compiler should be optimized in such a way that it can treat warnings and hints independently. With that optimization we expect a speedup for each kernel will be obtained.

B. Energy

IBM has not revealed much information about the SPE power consumption, but an estimation can be made for the 3.2 GHz SPE manufactured in the 90nm SOI process. Flachs et al. [8] made a Voltage/Frequency Schmoo that gives a power estimation for different frequencies and voltages. A 65nm SPE operates with Vdd= 0.9V [11]. The 90nm SPE uses a 100mV higher voltage [12], thus 1.0V. For these values, the Schmoo gives a power of 3W.

We use CACTI 5.3 [13] to create an estimation of the BTB's power consumption. CACTI is a tool for modelling dynamic and leakage power, area, and access time of caches and memories. The BTB is quite similar to a direct mapped cache. However, CACTI only supports caches with at least 8 bytes of data per line while the BTB has only 18 bits of data. Therefore, we use the method presented by Kahn [4] to correct this by scaling the word and bit line power in the data array with that factor of 18/64.

Table II shows the most important CACTI results, corrected for clock frequency and data size. The dynamic power assumes the BTB is accessed every cycle. However, in our case the BTB is only accessed for branch instructions. In the worst case (QuickSort), a branch instruction is executed in 5.07% of the cycles. For each branch instruction, the BTB is read and written. With leakage power added, the total power consumption of the BTB is 5.07% x (26.21 + 19.68) + 0.50 = 2.82mW, which is about 0.1% of the total SPE power consumption.

Table II
CACTI RESULTS FOR 256 ENTRY BTB, CORRECTED FOR DATA SIZE AND FREQUENCY.

Parameter	Value
Dynamic Read Power	26.21 mW
Dynamic Write Power	19.68 mW
Standby leakage per bank	0.50 mW

```
1  hbra    0x18,0x3e18
2  lqx     $2,$10,$19
3  rotqby  $2,$2,$8
4  and     $13,$2,$11
5  clgt    $3,$13,$25
6  brz     $3,0x8
7  brnz    $9,0x3ff94
```

Listing 1. Original MiniGZip code with hint.

```
1  hbra    0x24,0
2  lqx     $5,$5,$18
3  nop     $127
4  nop     $127
5  nop     $127
6  nop     $127
7  rotqby  $2,$5,$8
8  and     $12,$2,$10
9  clgt    $3,$12,$24
10 brz     $3,0x8
11 brnz    $15,0x3ff84
```

Listing 2. MiniGZip code with branch warning instead of hint.

Besides the BTB, the SBP also needs power for pre-decoding the instructions in the ILB. However, this is quite simple logic, thus we assume that it does not use more energy than a BTB lookup. The pre-decoding is performed every cycle and thus the additional power needed for the SBP predictors will be 1% or less. The branch warning predictor does not have to pre-decode instructions, but it has to prefetch the target instructions. Therefore the extra power is estimated to be 0.2%.

To determine the energy efficiency of the predictors, we calculate the total energy used for executing a program. As depicted in Table III, the speedup of all predictors is higher than the power consumption of the predictor. Therefore, the total energy decreases. The SBP has only a small improvement, due to its low speedup. The SBP with overruled hints (SBP-OH-NLS) has the highest decrease, because it has the best performance. Although the branch warning predictor (BWP-OH-NLS) has lower power consumption, the total energy is decreased less because of its lower performance. However, if an optimized compiler is used, the performance will improve and we expect it to have the largest energy reduction of all.

Table III
ENERGY EFFICIENCY OF THE BRANCH PREDICTORS IN TERMS OF TOTAL CONSUMED ENERGY OF THE ENTIRE APPLICATION.

Predictor	**Power increase**	**Average Speedup**	**Energy decrease**
SBP	1.0%	1.4%	0.4%
SBP-OH-NLS	1.0%	7.3%	6.2%
BWP-OH-NLS	0.2%	4.7%	4.5%

VI. CONCLUSION

In order to improve the branch prediction capabilities of the Cell SPE we proposed three hardware branch predictors. Predictions are calculated using a bimodal counter and are stored in a branch target buffer. The Simple Bimodal Predictor pre-decodes instructions while they are in the ILB, which makes it possible to do a BTB lookup for branch instructions only, saving energy. It ignores hints though. We also implemented a version of the SBP that uses hints, but they are overruled if the predictor predicts strongly not taken. The third predictor uses branch warning instructions to inform the SPU of an upcoming branch. Hints are also used and can be overruled too. Furthermore, we implemented an aggressive branch predictor to investigate the maximum performance gain possible.

The ABP is the fastest, but because of its complexity it is no serious candidate for implementation in the SPE. The SBP with overruled hints is second fastest, with an average speedup of 7.3%. With only 1% additional power consumption, the total energy for executing a program is reduced by 6.2%. Therefore this is currently the best predictor. The branch warning predictor requires only 0.2% extra power, but in some cases the performance was lower than it could be, because we did not optimized the compiler. If an optimal compiler is used, it is expected to be even more efficient than the SBP with hints.

REFERENCES

[1] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev, "Matched SAMS Scheme: Supporting Multiple Stride Unaligned Vector Accesses with Multiple Memory Modules," Tech. Rep., 2008, CE-TR-2008-06. [Online]. Available: http://ce.et.tudelft.nl/publications.php

[2] C. Meenderinck and B. Juurlink, "Specialization of the Cell SPE for Media Applications," in *Proc. Int. Conf on Application-Specific Systems, Architectures and Processors*, 2009.

[3] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan, "Power Issues Related to Branch Prediction," in *Proc. Int. Symp. on High-Performance Computer Architecture*, 2002.

[4] R. Kahn and S. Weiss, "Thrifty BTB: A Comprehensive Solution for Dynamic Power Reduction in Branch Target Buffers," *Microprocessors & Microsystems*, vol. 32, no. 8, 2008.

[5] C. Yang and A. Orailoglu, "Power Efficient Branch Prediction through Early Identification of Branch Addresses," in *Proc. Int Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[6] D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang, "Branch Prediction on Demand: an Energy-Efficient Solution," in *Proc. Int. Symp. on Low power Electronics and Design*, 2003.

[7] 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 VLSI Journal*, vol. 38, no. 3, 2005.

[8] B. Flachs *et al.*, "Microarchitecture and Implementation of the Synergistic Processor in 65-nm and 90-nm SOI," *IBM Journal of Research and Development*, vol. 51, no. 5, 2007.

[9] F. Cabarcas, A. Rico, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator," in *XVIII Jornadas de Paralelismo*, 2006.

[10] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, 2006.

[11] D. T. Wang, "ISSCC 2008 Cell Processor update," *Real World Technologies*. [Online]. Available: http://www.realworldtech.com/page.cfm?ArticleID=RWT022508002434&p=2

[12] M. Riley *et al.*, "Implementation of the 65nm cell broadband engine," in *Proc. Custom Integrated Circuits Conference*, 2007.

[13] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Laboratories, Tech. Rep., 2008. [Online]. Available: http://www.hpl.hp.com/research/cacti/