Extending the Cell SPE with Energy Efficient Branch Prediction

Martijn Briejer[1], Cor Meenderinck[1], and Ben Juurlink[2]

[1] Delft University of Technology, Delft, the Netherlands
cor@ce.et.tudelft.nl
[2] Technische Universität Berlin, Berlin, Germany
juurlink@cs.tu-berlin.de

Abstract. Energy-efficient dynamic branch predictors are proposed for the Cell SPE, which normally depends on compiler-inserted hint instructions to predict branches. All designed schemes use a Branch Target Buffer (BTB) to store the branch target address and the prediction, which is computed using a bimodal counter. One prediction scheme predecodes instructions when they are fetched from the local store and accesses the BTB only for branch instructions, thereby saving power compared to conventional dynamic predictors that access the BTB for every instruction. In addition, several ways to leverage the existing hint instructions for the dynamic branch predictor are studied. We also introduce branch warning instructions which initiate branch prediction before the actual branch instruction is fetched. They allow fetching the instructions starting at the branch target and thus completely remove the branch penalty for correctly predicted branches. For a 256-entry BTB, a speedup of up to 18.8% is achieved. The power consumption of the branch prediction schemes is estimated at 1% or less of the total power dissipation of the SPE and the average energy-delay product is reduced by up to 6.2%.

1 Introduction

Since a few years there is a clear trend towards multicore processors to increase performance. At the same time, the power consumption has become a major design constraint. Performance should therefore be maximized while staying within the power budget. The Cell processor, which is still one of the most advanced multicores, was designed exactly with this in mind. It contains one general purpose core, the Power Processing Element (PPE), and eight specialized cores, the Synergistic Processing Elements (SPEs). Especially the latter were designed with minimal area and energy consumption in mind [1]. This was achieved, among others, by omitting a dynamic branch predictor. Instead it uses compiler-directed branch hint instructions that change the fetch behavior, i.e., instead of fetching the instruction after the branch, they fetch the instruction at the branch target.

To further improve the Cell's performance, researchers have proposed various modifications to the SPEs. Different parts of the SPE have been targeted, ranging

from the memory system [2] to the instruction set [3]. However, to the best of our knowledge, no research has been performed to improve the branch prediction accuracy of the SPE.

Therefore, in this work we propose three dynamic branch prediction schemes for the SPE. Energy efficiency is obtained by minimizing the number of predictions, i.e., by avoiding table lookups. The first branch predictor scheme is a Simple Bimodal Predictor (SBP). In contrast to normal branch predictors, it only predicts when a branch instruction enters the pipeline. Despite early branch identification, it incurs a seven cycle branch delay. The second combines the SBP with hint instructions and thus can have a zero branch delay. The third dynamic branch predictor employs branch warning instructions to produce a prediction before the branch instruction is fetched, thereby obtaining a zero branch delay.

This paper is organized as follows. Section 2 provides a brief overview of the SPE architecture. Section 3 describes the experimental environment. In Section 4 the proposed dynamic branch predictors are presented. The performance and energy efficiency results are presented in Section 5. Related work is discussed in Section 6. Finally, conclusions are drawn in Section 7.

2 The SPU Architecture

The Cell processor consists of one PPE (Power Processing Element) and eight SPEs (Synergistic Processing Elements) [4]. 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 receive tasks from the PPE 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 can access the LS directly, 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.

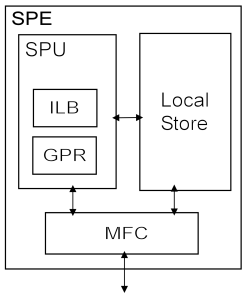

Fig. 1. Overview of the Cell SPE.

The SPU has 128 registers, each of which is 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 that a branch will be taken. If the hint is executed 16 cycles before the branch, execution can continue without delay if the hint is correct.

Despite the use 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 too close, only one of them is hinted. Second, hints are static and cannot change during the executing of a program while a branch predictor adjusts its prediction to the empirical findings. Finally, branches that are not hinted can be taken too. To improve the branching capabilities of the SPE we propose to extend it with a power-efficient dynamic branch predictor.

3 Experimental Setup

We used the CellSim [5] simulator to implement and test our branch predictors. CellSim is a modular simulator developed using the Unisim environment and very suitable for architectural research. It is an instruction set simulator, and therefore not all parts of the SPE are modelled cycle-accurate. However, its configuration parameters can be changed in order to obtain the performance close to actual performance. For all benchmarks used throughout this work, we tuned the configuration. The performance was validated using performance statistics from IBM SystemSim, which is a cycle-accurate simulator. The results of this performance validation are depicted in Table 1. It shows that in all cases the error is less than 5%.

Benchmarks from different application domains have been selected. 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 within 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

Table 1. Validation of CellSim against IBM SystemSim. Execution time 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%

representation of applications suitable for the Cell processor and comply with our requirements

MiniGZip is a parallel SPU implementation of the GZIP (de)compression program based on the ZLIB library, implemented by Seunghwa Kang [6]. The list ranking problem is a fundamental problem for many combinatorial and graph-theoretic applications. Bader et al. [7] developed an implementation for the Cell BE, which we optimized. The original benchmark is referred to as Listrank, while our optimized version is referred to as ListrankP3. MergeSort and QuickSort are the well known sorting algorithms. There are two different inputs, both containing 1024 elements. The first input is a decreasing sequence, while the second is a random input. When the latter is used, the name of the benchmark is extended with 'Rnd'. SPE-JPEG is a program made by Vitaly Vidmirov, that decodes a JPEG-image on the SPU. We used version 0.6 beta which can be downloaded from http://cellrb.blogspot.com/. A 512x384 demo image is included, which is used in the benchmark. The Deblocking Filter is also from the media domain. It is one of the kernels from the H.264 video processing coder/decoder and an implementation was made by Azevedo et al. [8].

4 Energy Efficient Prediction Schemes

In this section we describe the different branch predictor implementations. They are all based on a Bimodal Branch Predictor (BBP), which uses a bimodal counter to make a prediction. As depicted in Figure 2, the counter consists of two bits. The first bit indicates if a branch is predicted taken or not taken, while the second bit indicates if the prediction is strong or weak. When a branch instruction is executed, the counter is updated by adding one if it was taken or subtracting one if not. By default, the prediction is not taken.

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 address is also stored.

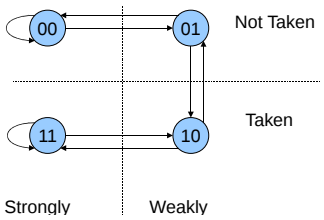

Fig. 2. State Diagram of the Bimodal Counter.

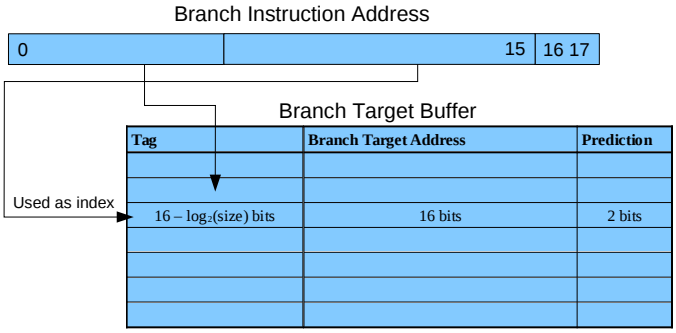

Fig. 3. Design of the Branch Target Buffer.

4.1 Simple Bimodal Predictor

The first implementation that uses the BBP is the Simple Bimodal Predictor (SBP). For energy efficiency, a BTB lookup is performed only for branch instructions. To be able to do that, the SPU needs to identify branch instructions. Normally, instructions have been decoded after stage 9 (ID2) of the SPU pipeline (see Figure 4), and thus in that stage the type of instruction is known. 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 (IB1) of the pipeline. In the next cycle, the prediction can be performed. 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 in the SBP predictor.

4.2 SBP Combined with Hints

The SBP ignores hint instructions, but they contain valuable information about the branch. If a (correct) hint is executed at least 16 cycles before the branch, the branch can be taken without delay, which is better than the seven cycle penalty of

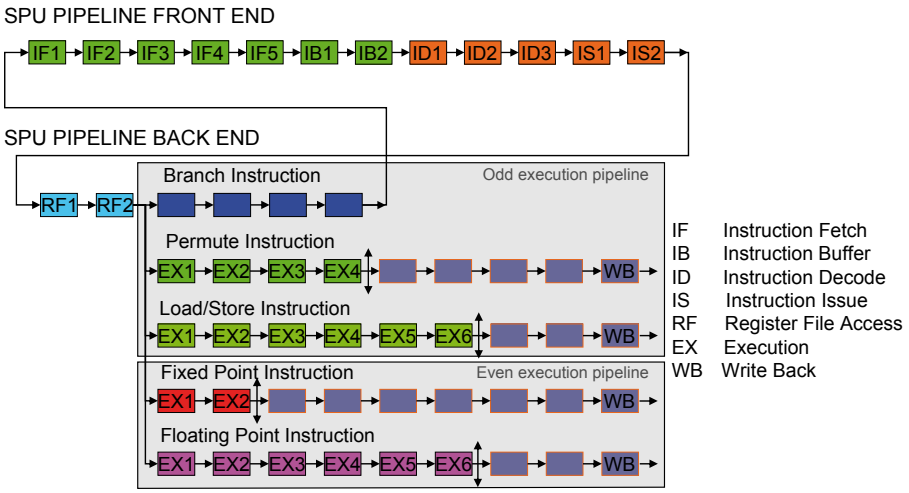

Fig. 4. SPU pipeline diagram (based on [9]).

the SBP. Therefore, we also made an implementation of the SBP that uses hints. Sometimes, however, hints are incorrect. Therefore, we implemented different hint policies, ranging from always using the hint to letting the predictor overrule the hint. We simulated the policies and the results showed that the policy that overrules the hint if the branch is strongly predicted not taken provides the highest performance. This branch prediction scheme is referred to as SBP-OH-NLS (SBP - Overrule Hints - Not Loading hint of Strongly not taken).

4.3 Branch Warning Predictor

To further improve performance, we want to perform the prediction earlier than the seventh cycle. Therefore, we introduce a new instruction: the branch warning. This instruction is similar to the hint instruction, except it is inserted for a branch with uncertain target. It is 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, provided the branch was correctly predicted and provided the branch warning was executed more than 16 cycles before the actual branch instruction. Hint instructions are also used, but are overruled when the predictor predicts strongly not taken. Thus the branch target buffer 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. The SPU is a dual issue processor, however, and therefore 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

(Branch Warning Predictor - Overrule Hints - Not Loading hint if Strongly not taken).

In our current implementation, the branch warnings are implemented as hint instructions with target 0. The compiler, however, assumes only one hint instruction can be active at any time. Therefore, the insertion of branch warnings interferes with hints. To illustrate this, the left side of Listing 1.1 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. The right side of the listing 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. (The nop instructions were added by the compiler to assure a distance of four instruction pairs between the hint and the corresponding branch instruction.) 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. Such a compiler optimization, however, has not been performed as of yet.

4.4 Aggressive Bimodal Predictor

To investigate how much the performance potentially can be improved by using a bimodal branch predictor, we also implemented an Aggressive Bimodal Predictor (ABP). The main difference with the previous predictors is that it performs a BTB lookup for every instruction. Because now there is no need to decode a branch before doing the prediction, the prediction can be performed 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 the pipeline when 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 SPE.

```
1  hbra    0x18,0x3e18          hbra    0x24,0
2  lqx     $2,$10,$19           lqx     $5,$5,$18
3  rotqby  $2,$2,$8             nop     $127
4  and     $13,$2,$11           nop     $127
5  clgt    $3,$13,$25           nop     $127
6  brz     $3,0x8               nop     $127
7  brnz    $9,0x3ff94           rotqby  $2,$5,$8
8                               and     $12,$2,$10
9                               clgt    $3,$12,$24
10                              brz     $3,0x8
11                              brnz    $15,0x3ff84
```

Listing 1.1. MiniGZip code. Left: Original with hint. Right: with branch warning instead of hint.

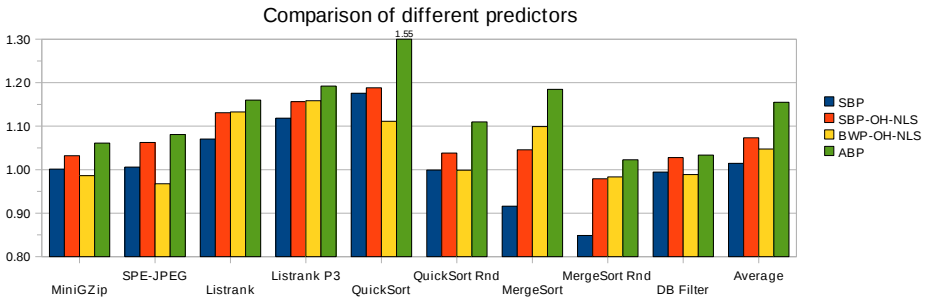

Fig. 5. Speedup, over the conventional SPE, obtained with the proposed branch predictors with a 256 entry BTB.

It was added to this analysis as a performance reference for the other branch predictors.

5 Evaluation

In this section the experimental results are presented. First, we evaluate the performance and then, we discuss the energy consumption.

5.1 Performance

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

As expected, the aggressive branch predictor provides the highest speedup for all benchmarks. On average, its speedup is 15.5%. For QuickSort the highest speedup is obtained, namely 55%. There are two reasons for that. First, about 45% of all cycles are 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 far ahead, and thus they achieve 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 only three benchmarks. In the original implementation, Listrank has a lot of wrongly hinted branches, while 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 for the

SBP 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). On average, however, the branch warning predictor has a speedup of 4.7%, which is in between both SBP predictors. The mixed results are due to the suboptimal algorithm the compiler uses to insert the branch warnings. Because of that, not all branches can be hinted/warned significantly affecting performance. We do expect though, that if an optimized compiler is used, the speedup of this branch warning predictor for all kernel will be larger and on average it exceeds that of the SBP-OH-NLS.

5.2 Energy Consumption

IBM has not revealed much information about the SPE's power consumption, but an estimation can be made for the 3.2 GHz SPE manufactured in the 90nm SOI process. Flachs et al. [4] present a voltage/frequency 'schmoo' that gives a power estimation for different frequencies and voltages. A 65nm SPE operates with Vdd= 0.9V [10]. The 90nm SPE uses a 100mV higher voltage [11], thus 1.0V. For these values, the schmoo yields a power of 3W.

To estimate the BTB's power consumption, CACTI 5.3 [12] was used. 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 [13] to correct for this by scaling the word and bit line power in the data array with that factor of 18/64.

Table 2 shows the most important CACTI results, corrected for clock frequency and data size. The dynamic power assumes the BTB is accessed every cycle. In our case, however, the BTB is only accessed for branch instructions. In the worst case, 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 $5.07\% \times (26.21 + 19.68) + 0.50 = 2.82$mW, which is about 0.1% of the total SPE power consumption.

Besides the BTB, the SBP also needs power for pre-decoding the instructions in the ILB. This logic is quite simple, however, and thus we assume that it does not consume more energy than a BTB read. This pre-decoding is performed every cycle and thus the total power consumed by the SBP predictor is 2.82 + 26.21 = 29.03 mW, which is 1% of the total SPEs power consumption. The branch warning predictor does not have to pre-decode instructions, but it has to prefetch the target instructions. We estimate the energy consumption of the latter to be

Table 2. CACTI results for a 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

equal to that of the BTB, and thus in total the BWPs power consumption 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 3, the energy consumption decreases for all branch predictors. Due to its large speedup, the SBP with overruled hints (SBP-OH-NLS) provides the highest improvement. Although the branch warning predictor (BWP-OH-NLS) has a lower power consumption, the total energy decrease is less because of its lower performance. However, using an optimized compiler, the performance will improve and is expected to have the largest energy reduction of all. The power consumption of the ABP was not calculated due to its complexity. Its energy efficiency is expected to be low.

Table 3. Energy efficiency of the branch predictors. Energy consumption is power times the execution time.

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%

6 Related Work

Several techniques have been proposed before to reduce the energy consumption of branch predictors, which can be up to 10% of the total energy consumption of modern processors [14]. Banking the branch predictor table reduces the active part when performing a lookup, thereby reducing the energy consumption. Parikh et al. [14] searched for an optimal banking strategy. They also proposed a Predictor Probe Detector (PPD), which pre-decodes instructions in the instruction cache to detect a branch. A BTB lookup is performed only for a branch, which reduces the energy consumption of the branch predictor by approximately 45%. Kahn and Weiss [13] reduced the number of BTB lookups by using a counting Bloom filter, that determines if an address is in the BTB or not. They reduced the dynamic power consumption by 51%. Yang and Orailoglu [15] 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. [16] used profiling information to adapt the predictor hardware on the fly. Monchiero et al. [17] 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.

Our Simple Bimodal Predictor (SBP) draws from the PPD in the sense that it also pre-decodes instructions to reduce the number of BTB lookups. PPD, however, pre-decodes all instructions in the instruction cache and stores the results in a table. We have incorporated the pre-decoding in the pipeline. Only instructions in the ILB are pre-decoded and no table is required. Further, we have extended the hint-based prediction scheme. Based on dynamically collected information, the statically determined hint can be overruled to optimize the prediction accuracy.

7 Conclusion

In order to improve the performance and reduce the energy consumption of the Cell SPE we proposed three dynamic branch predictors. Predictions are calculated using a bimodal counter and stored in a branch target buffer. The Simple Bimodal Predictor pre-decodes instructions while they are in the ILB, which makes it possible to perform a BTB lookup for branch instructions only, thereby saving energy. The SBP ignores hints though. We also proposed a version of the SBP that uses hints, but they are overruled if the predictor strongly predicts not taken. The third predictor uses branch warning instructions to inform the SPE 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 and power consumption it is not a 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 average total energy consumption 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 optimize the compiler. If an optimal compiler is used, it is expected to be even more efficient than the SBP with hints.

References

1. Hofstee, H.: Power Efficient Processor Architecture and the Cell Processor. In: Proc. Int. Symp. on High-Performance Computer Architecture (HPCA). (2005)
2. Gou, C., Kuzmanov, G., Gaydadjiev, G.N.: Sams: Single-affiliation multiple-stride parallel memory scheme. In: Proc. Workshop on Memory Access on Future Processors: a Solved Problem? (2008)

3. Meenderinck, C., Juurlink, B.: Specialization of the Cell SPE for Media Applications. In: Proc. Int. Conf on Application-Specific Systems, Architectures and Processors. (2009)
4. Flachs, B., et al.: Microarchitecture and Implementation of the Synergistic Processor in 65-nm and 90-nm SOI. IBM Journal of Research and Development **51**(5) (2007)
5. Cabarcas, F., Rico, A., Rodenas, D., Martorell, X., Ramirez, A., Ayguade, E.: CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator. In: XVIII Jornadas de Paralelismo. (2006)
6. Bader, D., Agarwal, V., Madduri, K., Kang, S.: High Performance Combinatorial Algorithm Design on the Cell Broadband Engine processor. Parallel Computing **33**(10-11) (2007)
7. Bader, D., Agarwal, V., Madduri, K.: On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In: Proc. IEEE/ACM Int. Parallel and Distributed Processing Symp. (2007)
8. Azevedo, A., Meenderinck, C., Juurlink, B., Alvarez, M., Ramirez, A.: Analysis of Video Filtering on the Cell Processor. In: Proc. Int. Symp. on Circuits and Systems. (2008)
9. Gschwind, M., Hofstee, H., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture. IEEE Micro **26**(2) (2006)
10. Wang, D.T.: ISSCC 2008 Cell Processor update. Real World Technologies
11. Riley, M., et al.: Implementation of the 65nm Cell Broadband Engine. In: Proc. Custom Integrated Circuits Conference. (2007)
12. Thoziyoor, S., Muralimanohar, N., Ahn, J.H., Jouppi, N.P.: CACTI 5.1. Technical report, HP Laboratories (2008)
13. Kahn, R., Weiss, S.: Thrifty BTB: A Comprehensive Solution for Dynamic Power Reduction in Branch Target Buffers. Microprocessors & Microsystems **32**(8) (2008)
14. Parikh, D., Skadron, K., Zhang, Y., Barcella, M., Stan, M.: Power Issues Related to Branch Prediction. In: Proc. Int. Symp. on High-Performance Computer Architecture. (2002)
15. Yang, C., Orailoglu, A.: Power Efficient Branch Prediction through Early Identification of Branch Addresses. In: Proc. Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems. (2006)
16. Chaver, D., Pinuel, L., Prieto, M., Tirado, F., Huang, M.: Branch Prediction on Demand: an Energy-Efficient Solution. In: Proc. Int. Symp. on Low power Electronics and Design. (2003)
17. Monchiero, M., Palermo, G., Sami, M., Silvano, C., Zaccaria, V., Zafalon, R.: Low-Power Branch Prediction Techniques for VLIW Architectures: a Compiler-Hints Based Approach. Integration VLSI Journal **38**(3) (2005)