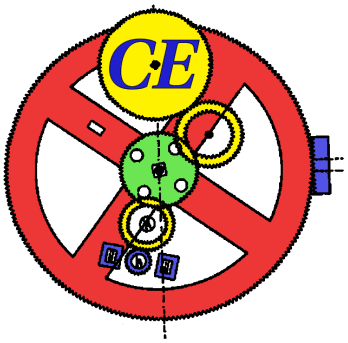

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Energy Efficient Branch Prediction for the Cell BE SPU

Martijn Briejer

CE-MS-2009-10

Abstract

We propose a power efficient branch predictor for the Cell SPU, which normally depends on compiler inserted hint instructions to predict taken branches. We designed four predictors all using Branch History Table (BHT) to store the Branch Target Address and the prediction, which is computed using a bimodal counter. The Simple Bimodal Predictor (SBP) predecodes instructions in the Instruction Line Buffer and accesses the BHT only for a branch instruction and ignores hints. For the second design, four ways to combine hints with the SBP are studied, by not or partially overruling hints by the predictor. We also introduce Branch Warnings (BW). The SPU only accesses the predictor when a BW or a hint is executed and hints can be overruled. The Aggressive Bimodal Predictor is an aggressive implementation of the SBP that starts predicting when instructions are fetched from local store. It is not designed for energy efficiency but to investigate the maximum possible speedup for a branch predictor not using hints.
Results show that a SBP in combination with overruling hints (SBP-OH-NLS) and a 256-entry BHT can have a speedup of 18.8%. The Branch Warning predictor is the fastest in some occasions, however a non optimal compiler makes it the worst for others. The SBP has the lowest performance overall. The estimated extra power needed for the SBP and SBP-OH-NLS is about 1% of the total SPE power and even less for the BW-OH-NLS. The energy-delay product is reduced the most for the SBP-OH-NLS.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Energy Efficient Branch Prediction for the Cell BE SPU

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Martijn Briejer
born in Voorburg, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Energy Efficient Branch Prediction for the Cell BE SPU

by Martijn Briejer

Abstract

We propose a power efficient branch predictor for the Cell SPU, which normally depends on compiler inserted hint instructions to predict taken branches. We designed four predictors all using Branch History Table (BHT) to store the Branch Target Address and the prediction, which is computed using a bimodal counter. The Simple Bimodal Predictor (SBP) predecodes instructions in the Instruction Line Buffer and accesses the BHT only for a branch instruction and ignores hints. For the second design, four ways to combine hints with the SBP are studied, by not or partially overruling hints by the predictor. We also introduce Branch Warnings (BW). The SPU only accesses the predictor when a BW or a hint is executed and hints can be overruled. The Aggressive Bimodal Predictor is an aggressive implementation of the SBP that starts predicting when instructions are fetched from local store. It is not designed for energy efficiency but to investigate the maximum possible speedup for a branch predictor not using hints.

Results show that a SBP in combination with overruling hints (SBP-OH-NLS) and a 256-entry BHT can have a speedup of 18.8%. The Branch Warning predictor is the fastest in some occasions, however a non optimal compiler makes it the worst for others. The SBP has the lowest performance overall. The estimated extra power needed for the SBP and SBP-OH-NLS is about 1% of the total SPE power and even less for the BW-OH-NLS. The energy-delay product is reduced the most for the SBP-OH-NLS.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-10

Committee Members :

Advisor:	Ben Juurlink, CE, TU Delft
Chairperson:	Kees Goossens, CE, TU Delft
Member:	René van Leuken, CAS, TU Delft
Member:	Cor Meenderinck, CE, TU Delft

This thesis is dedicated to my parents,
for their love and continuous support

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my supervisor Ben Juurlink for guiding me through my Master Thesis. In addition to this, I want to thank him for useful advice all along this project.

I also want to thank Cor Meenderinck for the valuable input he gave during the project by asking critical questions, assistance with the simulator and for proofreading this thesis.

I would like to thank my friends and family for supporting me and showing a lot of faith in me during my entire study. I especially would like to thank my fellow student Javier Quintela for all the great discussions and fun we had.

Last, but not least, I want to thank Kees Goossens and René van Leuken for being part of my graduation committee.

Martijn Briejer
Delft, The Netherlands
June 23, 2009

1 Introduction

The first section presents the motivation behind the presented work. In the second section we explain the objectives of this project. In the final section the organization of this thesis is described.

1.1 Motivation

Recently, there is a clear trend towards multi-core in order to increase the performance of modern processors. Because the size of transistors is decreasing with every new production technology, it is possible for manufacturers to put more transistors on the same area, which enables the use of multiple cores. Due to the power constrains however, the processor cores need to be more energy efficient. One of the most advanced multi-core processors is the Cell Broadband Engine (Cell BE) designed by IBM, Sony and Toshiba. Because of its area and energy efficient design, it contains one general purpose core, the Power Processing Element (PPU), and eight Synergistic Processing Elements (SPEs) which are SIMD media accelerators.

To improve the Cells performance further, researchers have proposed various modifications for the SPUs. Different parts of the SPU were targeted, from the memory system [1] to the instruction set [2]. Giorgi et al [3] added some extra hardware for thread scheduling and synchronization. However, no research was done to investigate the possible performance increase of hardware branch prediction.

The SPU has no dynamic hardware branch predictor and uses hint-for-branch instructions, inserted by the compiler, to statically predict branches. Optimized SPU code generally avoids branches by using select statements and loop unrolling. However, this is not possible for all code sequences and a branch predictor could improve performance. The Cell SPU was designed for high frequency and low power, thus a branch predictor should be energy efficient. Although various research is done on energy efficient branch prediction for different architectures, none of them targeted the SPU.

We propose four branch predictions. We tried to reduce the energy consumption by reducing the number of predictions. The Simple Bimodal Predictor (SBP) does a prediction only for branch instructions. 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. We also implemented an aggressive branch predictor as a reference, that does a prediction for each instruction when it is fetched from the local store.

Results show that the SBP in combination with overruling hints and 256-entry branch history table can have a speedup of 18.8%. The branch warning predictor is fast in some

occasions, however a non-optimal compiler makes it the worst for others. The aggressive predictor is fastest for all benchmarks, but the second best predictor is not far behind in most cases. We estimated that the energy consumption of the predictors is relatively low, thus the total energy consumption for running a program actually decreases.

1.2 Thesis Objectives

When we started this project our goal was to find some dwarfs and use them to investigate the scalability limitations of the Cell BE. Dwarfs are benchmarks that comply to one of the 13 communication and computation patterns defined by researchers from the University of California at Berkeley [4] to investigate the performance of multi-core systems. These 13 patterns belong to classes of important applications.

However, we soon found out that the limited branch prediction mechanism of the Cell BE SPU has a significant impact on the performance of many applications. Thus we refocused to energy efficient branch prediction for the Cell BE SPU with the following goals:

- Investigate if performance can be improved by adding a dynamic branch predictor to the Cell SPU.
- Propose techniques for power efficient branch prediction.
- Implement the proposed techniques in CellSim

1.3 Thesis Organization and Contributions

The remainder of this thesis is organized as follows. Related work on power efficient branch prediction is presented in Chapter 2. In Chapter 3 background information on the Cell BE is presented. Chapter 4 describes the current branch prediction capabilities of the Cell SPU and its limitations. We propose three energy efficient branch predictors and one more aggressive predictor as indication of the performance improvement that can be achieved by adding a branch predictor. Chapter 5 describes the two simulators we used for our experiments. IBM SystemSim was used as reference, while CellSim was extended with our branch predictors. The benchmark programs are described in Chapter 6. The process of porting the benchmarks to CellSim and finding a valid configuration is also described. Finally, the performance and energy efficiency results of the proposed branch predictors are presented in Chapter 7. Our conclusions are written in Chapter 8, together with recommendations for future work.

2 Related Work

A branch prediction unit uses energy to predict the outcome of branches. In modern architectures, the branch predictor power consumption can be up to 10% of the entire processor power consumption [5]. To save energy, the branch prediction unit can be made more energy efficient, but when the accuracy decreases, the total energy consumption decreases less, or might even increase. So the method used to save energy in the branch prediction unit should not decrease the prediction accuracy much. Many researchers have proposed techniques for energy efficient branch prediction for various architectures.

A branch predictor table can be split into several parts called banks. To read a prediction from the table, only the bank that contains it has to be active, which reduces the energy needed. Parikh et al. [5] searched for an optimal banking strategy. Also, a Predictor Probe Detector (PPD) is proposed, which pre-decodes the instructions in the instruction cache to detect a branch. This information is stored in an extra so-called pre-decode bit in the instruction cache for each instruction. Only when the PPD indicates a branch, a lookup to the predictor and/or Branch Target Buffer (BTB) occurs. This reduces the branch predictor energy consumption by approximately 45%.

Yang [6] proposed a Branch Identification Unit (BIU), to early identify incoming branch addresses using statically extracted program control flow information and branch distance. This information is loaded at the start of a basic block. Only when a branch occurs that is predicted taken by the BIU, a BTB lookup is performed. Also predictor entries are hibernated to reduce leakage power.

Modern processors use multiple predictors. The Branch Predictor Prediction (BPP) [7] selectively turns off 2 of the 3 used predictors, using the same predictor as used the last time the branch was executed. This saves 28% for non-banked and 14% for banked predictors.

Selective Prediction Access (SEPAS) [8] identifies well behaving branches and stores them. Using this information, the predictor and the BTB do not need to be accessed for every branch, avoiding unnecessary BTB lookups and predictor updates. Also, if possible only one predictor is used as with the BPP. PABU [9] is a slightly different version of SEPAS.

The branch predictor hardware can also be adapted on the fly. In [10] programs are divided into modules. Using profiling, the optimal configuration of the BTB size and the prediction algorithm is determined. The compiler adds configuration instructions in front of the module providing on the fly reconfiguration.

Like in the SPU, hint instructions can be used. These hint instructions [11] are inserted well in front of the branch instruction and inform the processor that a branch is coming. When the target and direction are known at compile time, they are included in the hint-instruction. Then the branch can be predicted right even without using the

branch predictor and the BTB. This technique is implemented on a VLIW processor, and the hint instructions are inserted only instead of NOP instructions and thus cause no overhead.

Reducing the number of bits in the BTB can save about 35% of power used by the BTB, as shown in [12]. The tag length can be reduced by half a byte and the target address length by one byte, without loss of performance. Besides saving power, silicon area and access time are also reduced.

Kahn et al. [13] reduce the dynamic power dissipation of the BTB by reducing the number of lookups. They propose two mechanisms: the serial-BTB and the filter-BTB. For the serial-BTB, the BTB is split in a tag array and data array containing the branch target. Only if the tag has a match, a look-up of the data array is performed. The filter-BTB uses a counting Bloom filter to determine if an address is in the BTB and thus a look-up is necessary. Because both methods introduce an extra delay, a small direct mapped BTB is also used in parallel, for a fast response in case of a hit. This gives a 51% reduction in dynamic power consumption at a cost of 1.2% less performance.

The techniques described above are for various architectures, although not for the Cell SPU that we target. We use some of the ideas and implemented them with some modifications for the Cell SPU. First, our Simple Bimodal Predictor pre-decodes instructions to detect a branch like the PPD. However, we do not use a separate table to store the results but incorporated this in the instruction fetch pipeline which saves area and time. We also combined this with hint instructions, which has not been done before.

Second, we also use the idea of inserting hint instructions for branches with unknown target (at compile time) to access the branch predictor. However, our so-called Branch Warning Predictor also does a prediction for hint instructions with known target address and can overrule it when the predictor predicts 'not taken'. This leads to more lookups, but the accuracy improves more.

3 Background: The Cell BE

In this chapter we provide some background information on the Cell BE which is investigated in this project.

The Cell Broadband Engine (Cell BE), developed by Sony, Toshiba and IBM is a heterogeneous shared memory multi-core processor [14]. It is originally designed for high performance with multimedia applications and games. Using two different core designs on one chip, each core design can be optimized for the task it performs.

The Cell BE consist of one POWER Processing Element (PPE) and 8 Synergistic Processing Elements (SPE), which are connected to each other and to the main memory by the Element Interconnect Bus (EIB) (see Figure 3.1). The PPE consists of a POWER Processing Unit (PPU) and Level 1 and Level 2 caches. The PPU is based on the 64-bit Power Architecture with vector media extensions (Altivec) and provides common system functions. The PPE runs the operating system and coordinates the SPEs.

The SPEs are designed to accelerate media and streaming workloads. The design was optimized for performance, power, and area. Area and power consumption were important design parameters. Less area means that more SPEs can fit in one core and less power makes higher clock frequency possible.

The SPE consists of a Synergistic Processing Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC) (see Figure 3.2). The 256 KB local store contains both

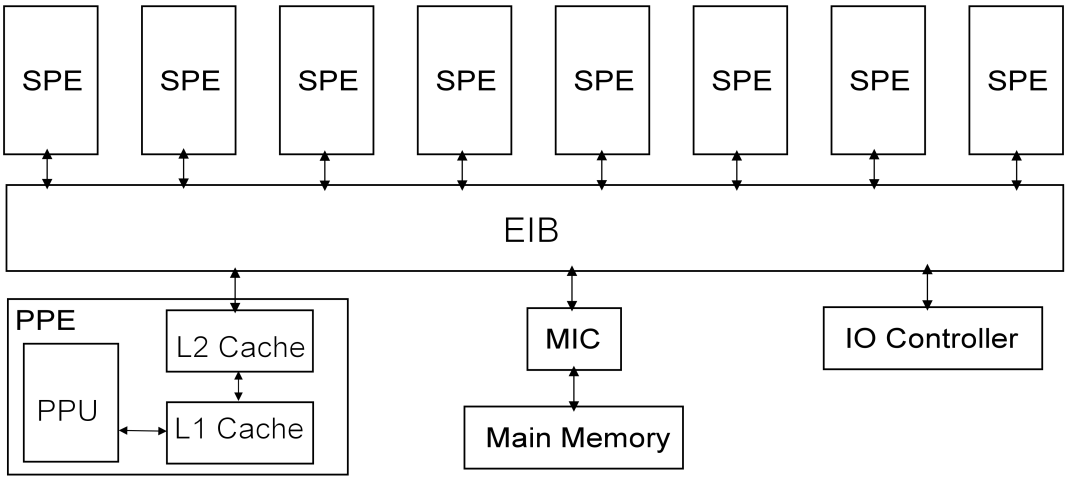

Figure 3.1: Overview of the Cell BE.

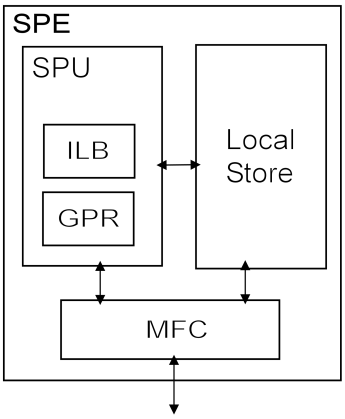

Figure 3.2: Overview of the SPE.

instructions and data and can be accessed directly by the SPU. The SPU can access the main memory through the MFC only using DMA Requests. The MFC then moves data between the local store and main memory using the EIB. It can also be used to communicate with the other SPUs and the PPU.

The SPU is a 32-bit RISC like processor. It uses instructions with a fixed length of 32 bits. The SPU has 128 128-bit general purpose registers. These can be used by both integer and floating point instructions. Because all SPU instructions are Single Instruction Multiple Data (SIMD) instructions, these 128 bit vectors can be treated as one quad-word, 16 bytes, and everything in between. The large number of registers allow loop unrolling and function inlining, which improves performance.

Instructions are fetched from the LS into the Instruction Line Buffer (ILB) in groups of 32 4-byte instructions called a line. The ILB can store up to 3.5 lines. The SPU pipeline issues instructions from half a line. Two lines are used for inline prefetching. The remaining line is used as a branch target buffer as explained further on.

The SPU can execute up to two instructions per cycle. The SPU has six functional units divided among an odd and an even execution pipeline (see Figure 3.3). Instructions are fetched from the ILB in pairs, called a fetch group. If two instructions from a fetch group are mapped to different pipelines, they can be executed simultaneously. To reduce dependency stalls, the SPU uses forward macros. This technique makes a result from an execution unit available as an input once it is calculated. Normally, the result can only be used as an input after it is written in the Register File.

Instructions are executed in order. The SPU has no dynamic hardware branch predictor. In case of branches, the SPU continues execution sequentially, except when a hint for branch instruction is used. Because of the long pipeline, a branch miss will result in a penalty of 18 cycles. A hint instruction (inserted at compile time) fetches the predicted branch target into the ILB. After the hinted branch instruction is fetched from the ILB, these instructions are fetched next. When a hint instruction is executed more than 16 cycles before the branch, the SPU can continue without delay if the branch is taken.

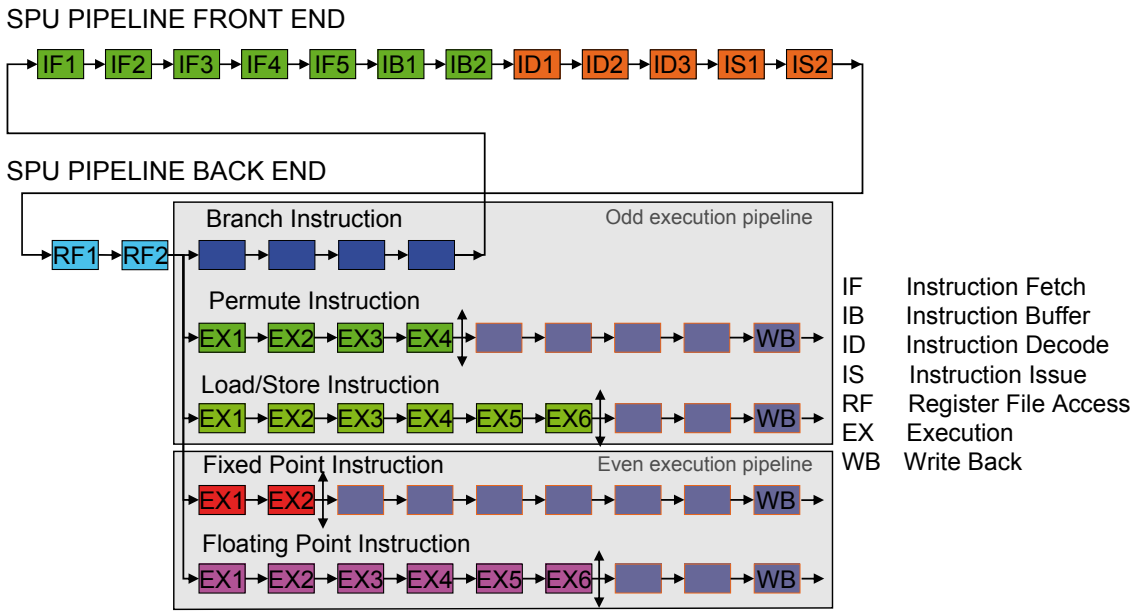

Figure 3.3: SPE Pipeline, taken from [15].

Energy Efficient Branch Prediction

4

The Cell SPU does not have a dynamic branch predictor. One of the reasons for that was to keep the power consumption low. However, the SPU does have some static branch prediction using hint for branch instructions. As will be explained in Section 4.1 this technique has some limitations. Code optimizations like loop-unrolling and changing if statements into select statements can be used to reduce the number of branches. However, in some cases this is not possible or effective.

In this chapter a few different dynamic branch predictors are proposed, which are designed for energy efficiency. They all use the same Bimodal Branch Predictor (BBP) described in Section 4.2, but in a different way. The Simple Bimodal Predictor (Section 4.3) uses only the BBP. The predictor in Section 4.4 uses the BBP in combination with hints. The Aggressive Branch Predictor in Section 4.5 is a non-energy-efficient implementation of the BBP to investigate how much performance improvement is possible. Finally, in Section 4.6 the hint instructions are extended with branch warning instructions in combination with the BBP.

4.1 Hint for Branch Instructions

The SPU uses hint-for-branch instructions to reduce branch miss stalls. Normally, the SPU predicts a branch 'not taken'. But a hint instruction lets the SPU predict 'taken', prefetches the target instructions and lets execution continue without delay when the branch is taken.

These hint instructions are inserted some distance (at least four instruction pairs and 11 cycles in order to branch without delay) [16] ahead of a branch instructions. The operands of a hint-for-branch instruction are the Branch Instruction Address (BIA) and the Branch Target Address (BTA). A hint instruction loads the SPU's Branch Target Buffer (BTB) with the BTA and the BIA. The BTB monitors the instruction stream going into the issue stage of the pipeline. When the instruction address matches the BIA, the SPU continues with the instructions from the BTA, which are prefetched into a special line of the Instruction Line Buffer (ILB).

When the BTB matches the BIA, the prefetched instructions are seamlessly put after the branch instruction in the pipeline. If the branch is taken, execution can continue without delay. If it is not taken (hint miss), the buffer is flushed and new instructions need to be fetched from local store. Then the normal branch penalty of 18 cycles applies.

If the hint is more than the four required instruction pairs in front of the branch instruction, but the other requirement of the 11 cycles is not met, the SPU goes into hint stall mode. The four instruction pairs are required, because a hint instruction needs four pipeline stages to execute and set the trigger. If the trigger is not set, the SPU does not know that there is a hint for the branch. After executing the hint, the branch

target is fetched. It takes at least 11 cycles to do this and put the target instruction in the pipeline just after the branch instruction. The branch instruction is stalled until this can be done, which is called Hint Stall.

4.1.1 Limitations

Using hint instructions for branch prediction has some limitations. First, it is static and not dynamic, thus it cannot adapt the prediction during the execution. The compiler uses some heuristics to determine if a branch is taken or not. For example, a loop-iteration branch is always predicted taken, while a loop-termination branch is predicted not taken. Or when a conditional branch depends on a comparison greater than zero it is predicted taken, while for less than zero it is predicted not taken. The actual values compared are not used in this prediction, only the program code.

The compiler tries to insert a hint instruction as far as possible (maximum 256 instructions) in front of the branch instruction. If two branches that should be hinted are less than four instruction pairs away, the hint cannot be inserted because there can be only one hint active at a time. Thus if the first branch is hinted, the second hint cannot be inserted before the first branch, because that would replace the first hint. The compiler can chose between hinting only one of them, or adding NOP instructions to increase the distance. The NOP instructions will incur extra performance cycles and most likely, the SPU still needs to go into Hint Stall mode.

Inserting hint instructions sufficiently in front to continue execution without delay can be difficult or even impossible. In some cases it is even impossible to insert a hint more than four instruction pairs in front. If insertion is succeeded, the branch prediction is based on static compile time heuristics. So branches that are hinted 'taken' are likely to have some delay, but it is also possible that the hint is wrong. And non-hinted branches (and thus predicted 'not taken') can also be taken.

4.2 The Basis: a Bimodal Branch Predictor

This section describes the Bimodal Branch Predictor, which will be used as the basis for the branch predictors we propose in the next sections.

We use a bimodal branch predictor because is a simple predictor. Other predictors like a local or a global predictor use more complex algorithms to do a prediction. They can be more accurate, but doing the prediction takes more time and energy. The used area is also larger.

Predictions are calculated with a 2-bit saturation counter. The prediction can have the following values:

Value	**Meaning**
00	Strongly Not Taken
01	Weakly Not Taken
10	Weakly Taken
11	Strongly Taken

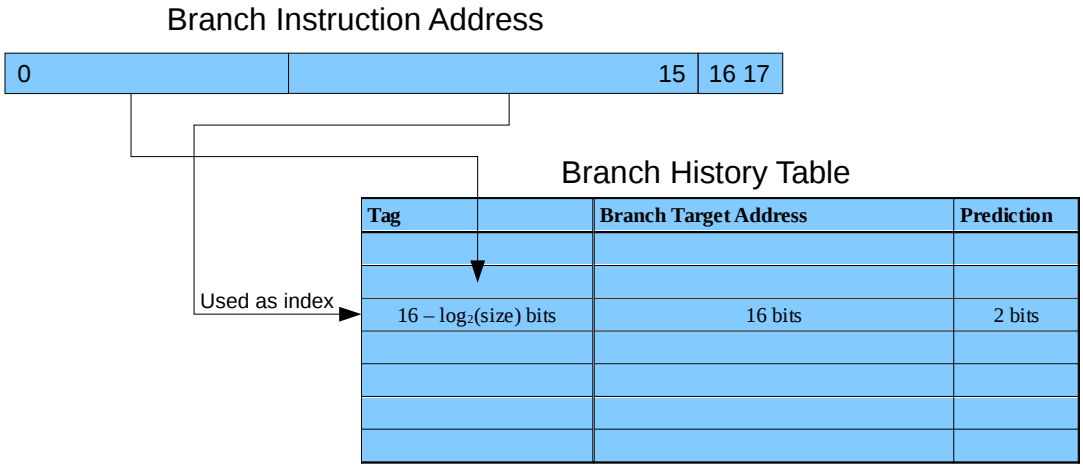

Figure 4.1: Design of the Branch History Table.

The leftmost bit determines if the branch is predicted 'taken' or 'not taken'. If a branch is taken, one is added to the prediction. If the branch is not taken, one is subtracted. When 00 or 11 is reached, the counter is saturated. Adding one to 11 or subtraction one from 00 does not change the value any more.

We implemented a branch predictor which uses a bimodal predictor scheme using a Branch History Table (BHT). The BHT has BHT_SIZE entries and each entry has three fields: a tag, the branch target address, and the prediction. It is indexed by the $\log_2$(BHT_SIZE) least significant bits of the 16-bit branch instruction word address (see Figure 4.1). Since instructions are stored in the Local Store, which is only 256 KB, there addresses can be represented with 18 bits. Because instructions addresses are word aligned, the two least significant bits are always zero and can be discarded. Therefore the tag should contain the remaining 16 - $\log_2$(BHT_SIZE) most significant bits of the branch instruction address. In case of BHT_SIZE = 256, the index is 8 bits and the tag is 8 bits.

The branch target address contains the predicted target of the branch instruction. When a branch is predicted taken, instructions are fetched from this address. 16 bits are sufficient to address the instruction words in the 256 KB Local Store.

It is assumed that a prediction can be done in one cycle. Agarwal et al [17] investigated the access times of various structures including a Branch Target Buffer (BTB), which structure is comparable to that of our BHT. This was done for design with a different clock cycle time, expressed in terms of the time required for an inverter to drive four copies of itself, called an fan-out-of-four (FO4). The access time for a 512 96-bit entry BTB is one cycle for a 16 FO4 design and two cycles for a 8 FO4 design, when 70nm or 100nm technology is used to make the structure. The Cell has a 11 FO4 design and is made with 90nm technology. Our BHT has as most 512 entries, but the entries

contain only 30 bits. This decreases the access time. Therefore it is reasonable to assume an access time of one cycle.

4.3 A Simple Bimodal Predictor

In this section the first energy efficient branch predictor is proposed, based on the BBP. The Simple Bimodal Predictor (SBP) only uses the BBP to do a prediction. Instructions are pre-decoded in order to do the prediction as early as possible. Hint instructions are ignored.

In this implementation, the BBP is accessed when an instruction is read from the Instruction Line Buffer and enters the SPU pipeline. For energy efficiency reasons, this is done only for branch instructions. This saves many unnecessary lookups for non-branch instructions, because branches are just a small fraction of the code.

To only access the BHT for a branch instruction, the predictor should know if the instruction is a branch or not. To do this, the instruction has to be decoded. The original SPU pipeline depicted in Figure 3.3 shows that there are three Instruction Decode (ID) stages. The instruction is decoded in stage ID1 and ID2 and stage ID3 is used for dependency checking. Thus in the ninth stage ID2, it is known if there is a branch instruction in the pipeline. The branch prediction can take place in the next cycle. Then a correctly predicted taken branch will have a penalty of 10 cycles instead of the normal 18 cycles. The modified SPU pipeline for the SBP is depicted in Figure 4.2.

Instead of waiting for the instruction to be decoded in the SPU pipeline, the branch detection is moved to the front. Like the Predictor Probe Detector [6], the instruction line buffer can be extended with some logic that (partially) pre-decodes the instruction to detect a branch instruction. When the instruction is fetched from the ILB, a extra bit indicates if it is a branch or not. In case of a branch, the BBP is accessed in the same cycle. The benefits of this is a smaller penalty. However, some extra logic to pre-decode/detect a branch instruction is necessary.

The branch prediction is done when a branch is fetched from the Instruction Buffer (stage IB2). A BHT lookup is done using the program counter. If the branch is predicted taken, its target instructions are fetched from local store. It takes 7 cycles before the first instruction can be read from the instruction buffer. Meanwhile the pipeline is fed with NOP instructions. So the penalty for a correctly predicted taken branch is 7 cycles (this was 18 cycles without prediction). It is assumed that a BHT lookup can be done in the same cycle as the instruction fetch.

Hint instructions go through the pipeline like normally. However, they are handled like a NOP instruction, thus they do not have any effect on branching and/or fetching instructions. Simply, instead of executing hint instructions, they are ignored.

The SBP was implemented in CellSim by changing the file spu.sim. The predictor can be enabled by setting the macro definition USE_PREDICTOR to 1 (0 for no predictor). Because hints are ignored, the macro definition USE_HINT must be set to 0.

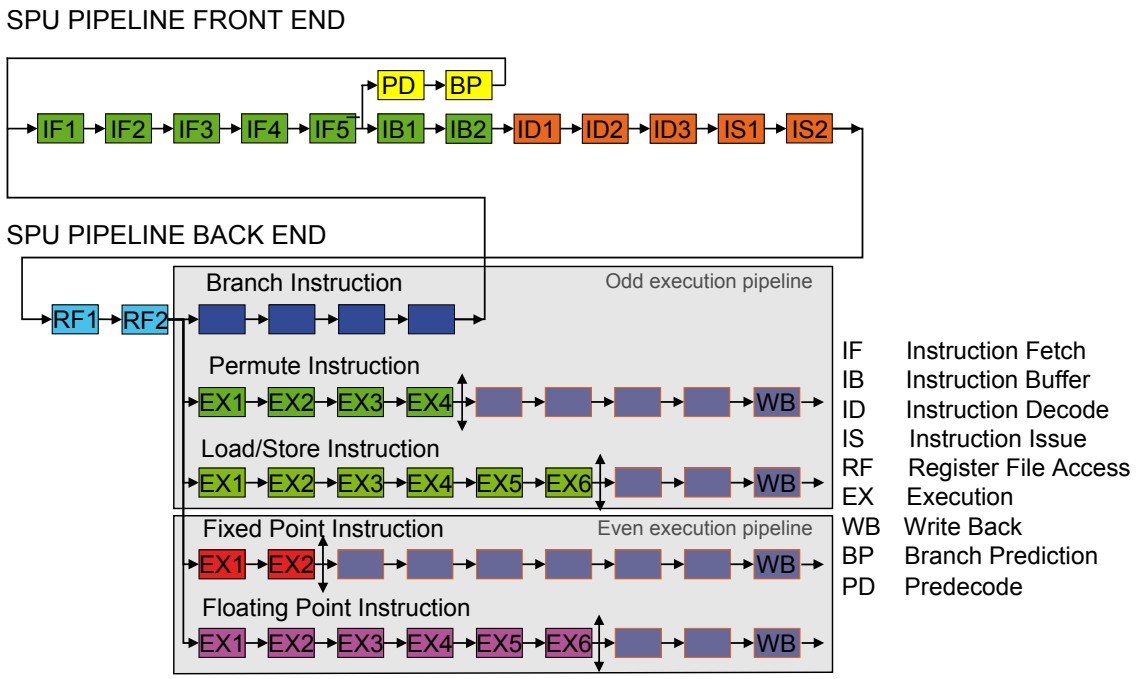

Figure 4.2: Modified SPU pipeline for the Simple Bimodal Predictor.

4.4 Combining the Simple Bimodal Predictor with Hints

In this section we propose a branch predictor that uses the BBP and hint instructions. We extend the SBP described in the previous section. The SBP uses only BBP, and ignores the hint instructions, including the valuable branch information in it. By using both dynamic branch prediction and hint instructions, we can take advantage of both.

The straightforward implementation is to use the branch hints for hinted branch instructions and to use the predictor for non-hinted branch instructions. But in part 3 of the ListrankMS SPU program (described in Section 6.1), only half of the hints are correct (see Table 6.6). So it could be more effective to let the branch predictor overrule a branch hint when it was wrong for the last couple of times.

Therefore we implemented a scheme where the predictor can overrule the branch hint when the prediction is opposite to the hint, thus not taken. This can be done only for strongly not taken predictions or also for weakly not taken predictions. These options are selected by setting the macro definition BP_OVERRULES_LIMIT in spu.sim to respectively 0 and 1. The macro definition BP_OVERRULES is used to control the overruling:

BP_OVERRULES	Behaviour
0	Hints are always used.
1	Hint targets are not fetched at all when branch is predicted not taken by the predictor.
2	Hint targets are fetched, but not used while the branch predictor predicts not taken.

We implemented the third option because some hints are used multiple times. For example, the hint instruction for a branch in a small loop can be inserted outside (thus before) the loop. The hint instruction is executed once, but the fetched hint target can be used every iteration, until they are replaced because another hint instruction is executed. If we do not fetch the hint target, this is not possible. Therefore it could be more effective to load the target, but not to use it.

We used four different combination of BP_OVERRULES_LIMIT and BP_OVERRULES for our research which are described in table 4.1. The modified SPU pipeline for these predictors is shown in Figure 4.3.

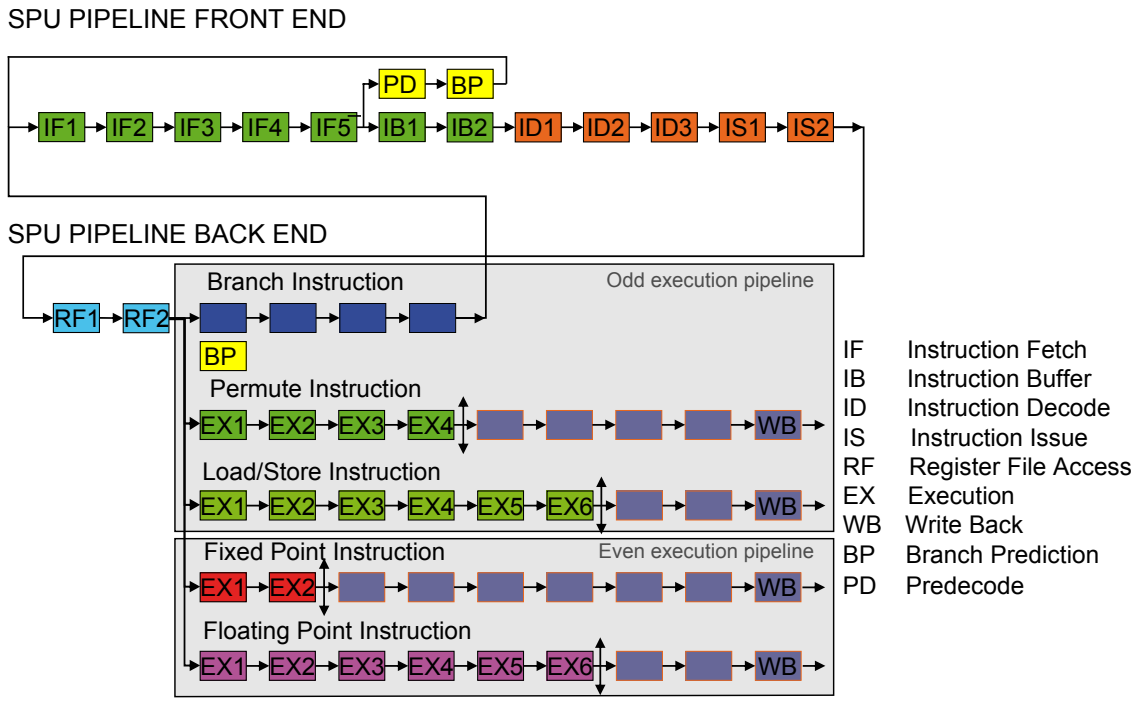

Figure 4.3: Modified SPU pipeline for the Simple Bimodal Predictor combined with hints.

Table 4.1: Different policies used to combine hints with the SBP.

Name	Description
Always	Hints are always used like normally. For non-hinted branches, the predictor is used.
OH-NLS	Hints target is not loaded when the predictor predicts Strongly Not Taken
OH-NUS	Hints target is loaded but not used while the predictor predicts Strongly Not Taken
OH-NLW	Hints target is not loaded when the predictor predicts Strongly or Weakly Not Taken

4.5 An Aggressive Branch Predictor

To investigate how much the performance can be improved by a bimodal branch predictor, we also implemented a more aggressive version of the bimodal predictor. We did not constrain ourselves to make a energy efficient or simple design. The most important feature is that this predictor does a prediction for every instruction, without knowing if it is a branch or not. In this way, the predictor can already start predicting in stage IF1, when the instructions are fetched from the local store to the ILB (after a flush) or are in the ILB. Doing a prediction for all instruction costs more energy than predicting

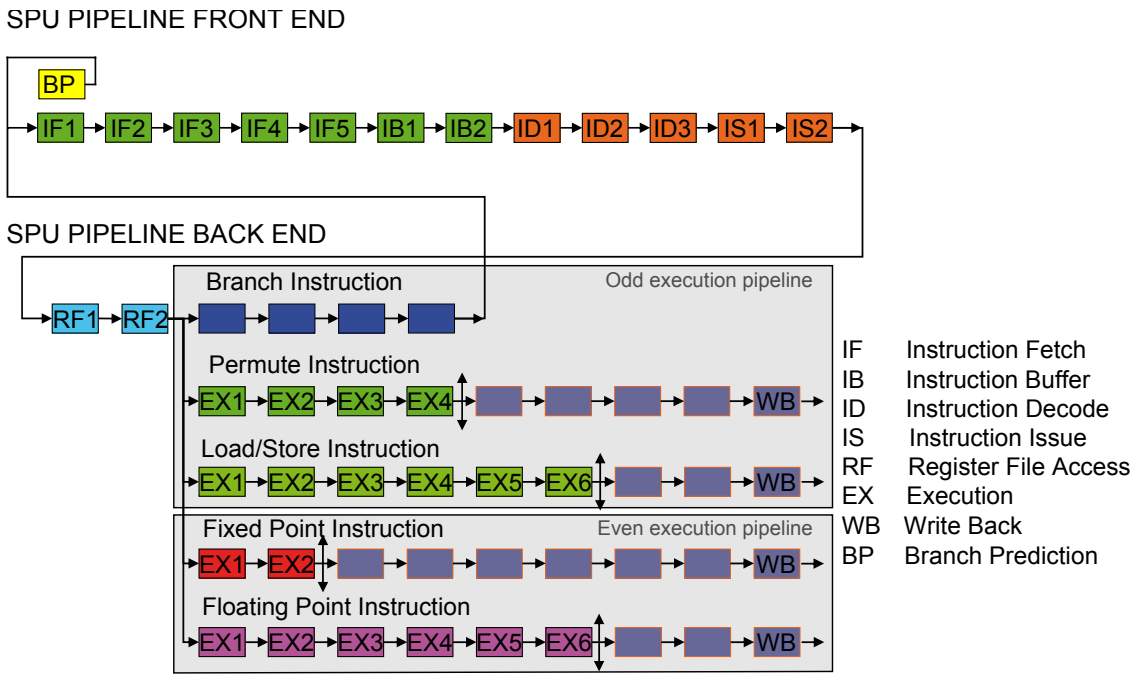

Figure 4.4: Modified SPU pipeline for the Aggressive Branch Predictor.

only branch instructions because a lot more BTH accesses are necessary.

When the ILB is empty, i.e., after a flush, new instructions are fetched from the local store. The PC of the first instruction is known at that time, so the predictor can be accessed. While the SPU is waiting for the instructions to be fetched, every cycle two predictions are done (dual issue). So when the first instruction is in stage ID1, the predictor has already been accessed 14 times. From now on, a prediction only is done when a instruction is executed. Thus there can be 0 (stall), 1 or 2 (dual issue) predictions per cycle. Figure 4.4 shows the modified SPU pipeline.

The branch predictor is 14 cycles ahead of the real execution. This is a speculative execution path and when the predictor is wrong, the ILB needs to be flushed the same way when a normal branch miss occurs. But when the predictor is right, there is no penalty for a branch.

There are two parameters that can be modified to change the behavior of this predictor. First, the macro definition PREDICTION_WIDTH in spu.sim defines the number of instructions to process while the SPU is fetching instructions from the local store after a flush, which is set to 14 in our experiments. Second, the macro definition PREDICTION_DEPTH in BranchPredictior.hxx defines the number of branches that can be taken speculatively by the predictor. As shown in Table 4.2 shows, most of the benchmarks (see Chapter 6) we use have less than eight outstanding taken branches. For the others, increasing PREDICTION_DEPTH above eight does not have a significant effect on the performance, thus we will use PREDICTION_DEPTH=8 in our experiments.

Table 4.2: Maximum number of outstanding taken branches using the ABP.

Parameter	**Value**
ListrankMS	4
QuickSort	17
QuickSort Rnd	10
MergeSort	4
MergeSort Rnd	4
MiniGZip	5
SPE-JPEG	5
Deblocking Filter	8

The instruction line buffer has to be extended in order to store the prefetched instructions with PREDICTION_DEPTH lines, one for each outstanding speculatively taken branch.

The Aggressive Branch Predictor is not a serious candidate for implementation in the SPU. The SPU is designed to be simple, energy efficient, and small. This predictor is more complex than the others, because it is much more speculative. It also is less efficient, because it does a prediction for every instruction fetched. And finally, the used area is twice as large, because of the eight extra lines in the ILB that cost about the same area as a 256 entry BHT.

4.6 A Predictor using Branch Warning Instructions

In Section 4.3 and 4.4, energy efficiency of the branch predictor is achieved by accessing the BHT only when a branch instruction is fetched. For this, it is necessary to know if the fetched instruction is a branch instruction. This can be done by accessing the predictor further down the pipeline, or pre-decode the instructions in the ILB. In this section we propose a new approach: using extra instructions to let the SPU know a branch is coming, which we will call *Branch Warnings*.

When a branch warning instruction is executed, the SPU uses the BBP to do a prediction for the branch the warning belongs to. For branches that are not preceded by branch warning, no prediction is done. When the branch warning is scheduled sufficiently far in front of the branch instruction (like for normal hint: four instruction pairs and 11 cycles), a correctly predicted branch has no penalty. Due to the double execution pipeline, these extra instructions could possibly be scheduled in such a way that they do not introduce extra execution cycles. However, the extra instructions need to be fetched from memory, which could cost extra time. Also, the size of the program will increase. This could be a problem for some programs, because of the limited size of the local store. This is also true for hints.

We implemented the branch warnings as hint-for-branch instructions with target 0. Therefore, the handling of hints with 0 is changed. Instead of loading the target, the BBP is accessed. When it predicts the branch taken, it fetches the branch target. The target is thus in a different (extra) line in the ILB than a normal hint. If there is a hint active, it stays active when a branch warning is executed. In case the branch is predicted not taken, the execution continues sequentially and no actions are taken.

When the branch instruction is executed, its target is compared to the predicted target. If they are the same, execution can continue. Else the pipeline is flushed and the correct target is fetched.

For hints, the SPU stalls when the target is not ready yet when it is needed. However, if the branch is not taken, thus the prediction is wrong, these stall cycles are an unnecessary extra penalty on top of the normal branch miss penalty of 18 cycles. Therefore we do not stall the SPU to wait for the Branch Warning target, but feed the pipeline with NOP instructions after the warned branch instruction. The execution of instructions in the pipeline continues and when the branch is not correctly predicted 'taken', there is no extra penalty. For correctly predicted branches, there is no performance difference.

The branch warnings are used in combination with hint instructions. The macro definition BP_OVERRULES determines the behavior of the hint instructions. If 0, the hints are always used. If 1, the hint instruction is ignored when the BBP predicts strongly not taken and thus the hint target is not fetched. The modified SPU pipeline of this predictor is depicted in Figure 4.5.

4.6.1 Compiler Modifications

To insert the branch warnings in the program, the compiler had to be modified. We modified GCC 4.1.1, the file gcc/config/spu.c. Hints are generated in the optimization pass of gcc, in the function *spu_machine_dependent_reorg_mb()*. For each branch

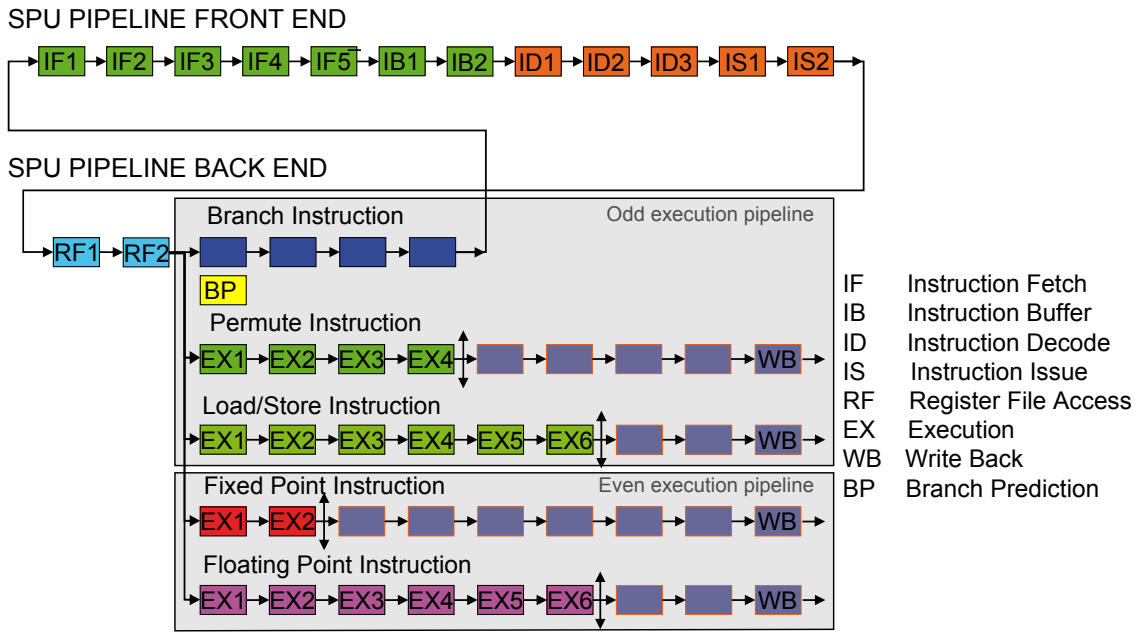

Figure 4.5: Modified SPU Pipeline for the Branch Warnings Predictor.

instruction, the compiler checks if it can determine the branch target using the function *get_branch_target()*. Originally, when the compiler could not determine if a branch was taken, 0 was returned. Then no hint for branch instruction was generated.

We replaced the return 0 statements in the function *get_branch_target()* to return GEN_INT(0). The function GEN_INT(0) creates an internal representation of an integer with value 0. GCC now thinks the branch has 0 as predicted target and will try to insert a hint for branch instruction in front of this branch. For branches of which a target can be determined, a normal hint is generated. Some branches will have no branch warning, because the compiler cannot place the instruction far enough ahead. The warning should be placed at least 4 instruction pairs before the branch. No other hint instruction can be placed between a hint and its branch.

The placement of branch warnings can be improved. Branch warnings are handled differently in hardware, therefore they can also be handled differently at compile time. They could be placed in between a hint and its branch, because they do not interfere in hardware. Probably, a new instruction should be added instead of using the hbra instruction with target 0. In that way they can be easily distinguished and thus scheduled differently. Currently, the compiler assumes that there can be only one hint or branch warning (because they are the same for the compiler) active at the same time. This means the compiler in some cases has to chose if it inserts a branch warning or a hint, while by using two different instructions they both can be inserted. This also gives other possibilities to improve the performance, for example by allowing two or more outstanding branch warnings by adding extra lines to the ILB. Because these compiler

changes need a thorough knowledge of GCC, this work has not been done but it would be very interesting to do in the future.

4.7 Summary

In this chapter the limited branch prediction capabilities of the SPU are discussed. We proposed four types of branch predictors to improve the branching behavior of the SPU. All predictors use a Bimodel Branch Predictor (BBP) as base, that uses a branch history table to store the prediction and the target address.

Only two of the proposed branch predictors use hint instructions and can overrule the hint if the predictor predicts not taken, while the others ignore the hint instructions. Because we defined four policies for using hint instructions (see Table 4.1), we have eight branch predictors to investigate. Table 4.3 shows the names we will use and the properties of those branch predictors.

The branch predictors have different Branch Hit Delays (BHD) because of the different pipeline stage the prediction is done. The branch miss penalty of 18 cycles is not changed by adding a branch predictor to the SPU. For energy efficiency, we tried to minimize the number of predictions. The ABP is the only branch predictor that does a prediction for every instruction that enters the pipeline, the other predictors do a prediction only for branch, hint or branch warning instructions.

Table 4.3: Properties of the different Branch Predictors. The abbreviation in the name column is how we will refer to this predictor from now on. Branch Hit Delay (BHD) is best case delay in cycles for a correctly predicted taken branch. BBP Access indicates when the predictor does a prediction. Hint policy indicates if and how the predictor uses the hint-for-branch instructions (see Table 4.1 for explanation of the policies)

Name	**Type of Branch Predictor**	**BHD**	**BBP Access**	**Hint Policy**
SBP	Simple Branch Predictor	7	Only for branches	None
SBP-H	SBP combined with hints	7	Only for branches and hints	Always
SBP-OH-NLS	SBP combined with hints	7	Only for branches and hints	OH NLS
SBP-OH-NUS	SBP combined with hints	7	Only for branches and hints	OH NUS
SBP-OH-NLW	SBP combined with hints	7	Only for branches and hints	OH NLW
ABP	Aggressive Branch Predictor	0	For every instruction	None
BWP-H	Branch Warning Predictor	0	Only for Branch Warnings and hints	Always
BWP-OH-NLS	Branch Warning Predictor	0	Only for Branch Warnings and hints	OH NLS

5 Experimental Environment

During our research, several development and experimental environments have been used, which are described in this chapter. Besides a hardware Cell Blade server (Section 5.1), we also used 2 simulators: The IBM SystemsSim (Section 5.2) provided with Cell SDK, and CellSim (Section 5.3) which we could extend with new branch prediction techniques. However, before we could extend CellSim, some modifications had to be made in order to make CellSim more accurate. These modifications are also described.

5.1 Cell Blade Server

Some of the tests are done on the Cell BladeServer from the Barcelona Supercomputing Center. This is a prototype blade server with two 3.2 GHz Cell BE processors having 8 SPUs and 512 MB main memory each. It runs Fedora Core 7 with Cell SDK 3.0 installed. Applications are compiled using the GNU toolchain 4.1.1.

5.2 IBM SystemSim

The Cell Software Development Kit 2.1 comes with a simulator: the IBM Full-System Simulator for the Cell Broadband Processor Version 2.1 (SystemSim) [18]. SystemSim is a sophisticated simulator that can be used for both functional and performance simulations of the Cell BE. Besides simulating the processor, other parts of the system like network, I/O devices and disks are simulated as well. This allows an operating system to run on the simulator, in this case Linux.

SystemSim was used for development and testing of applications. Performance statistics were also gathered using this simulator. This was done in default configuration, which means an 8 SPU Cell running at 3.2 GHz and 256 MB main memory. The simulated operating system is Fedora Core 5. Applications are compiled using the GNU toolchain 4.1.1.

5.3 CellSim

SystemSim is a fast simulator which is very suitable for developing applications. Although it is possible to change machine characteristics like main memory size, it is not possible to change the architecture of the simulated system without the source code that is not publicly available. This makes it not suitable for architecture research. Therefore, another simulator is used: CellSim [19]. This is a modular simulator based on UNISIM [20]. UNISIM takes care of the communication between the modules and executes the required functionality each cycle, also providing a general clock signal. Each

module is a C++ class with methods for communication between modules. In CellSim, the Cell BE is divided in the modules which are connected as shown in Figure 5.1. The PPE consist of two modules: a PPU, and a cache. The SPE consist of a SPU, a Local Store and the MFC. The interconnection network connects the PPU, SPU, and Main Memory.

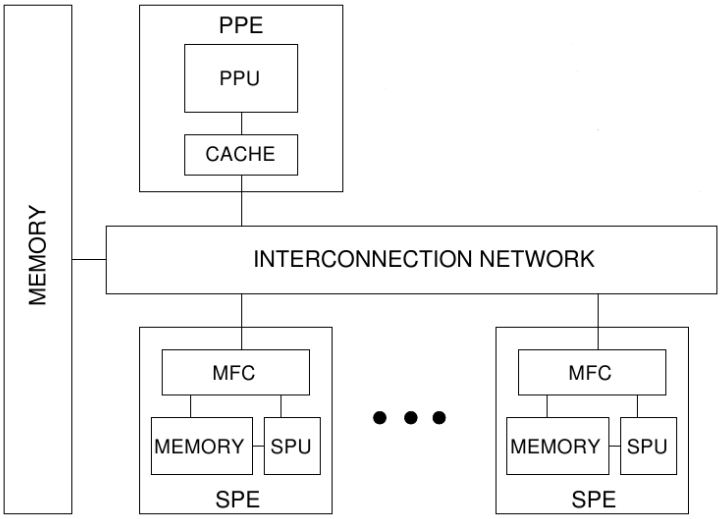

Figure 5.1: Block Diagram of CellSim modules, taken from [19].

We used CellSim version 0.9 in combination with UNISIM 3.5. For the PPU, memory, cache, and interconnection network the default configuration is used. The configuration changes made in the other modules are described in Chapter 6. Applications are compiled using the GNU toolchain 4.1.1.

5.3.1 Limitations

Unlike IBM SystemSim, CellSim is not a cycle accurate simulator but an instruction set simulator. This means that CellSim mimics the behavior of a Cell BE, but not every aspect is simulated correctly. For example, the pipeline stages of the SPU (which is the main subject of this research) are not simulated separately, but a instruction is executed in one simulation cycle. Latencies are used to stall the simulator when dependencies occur or when the SPU has to wait for load. However, CellSim has a lot of configuration parameters which tweak the performance of the simulator. With this, CellSim can be configured for each program in such a way that its performance is close to the real world. For each benchmark used in this research (Chapter 6), this validation is done.

Also, CellSim does not run an operating system, but only the program. As a result the simulator has to handle system calls, which is normally done by the operating system. However, not all system calls are implemented in CellSim. Programs that use unimplemented system calls will not work correct. These programs need to be modified so they do not use these system calls any more, which we did for some of the benchmarks

we used.

5.3.2 Bugfixing CellSim

When executing the first benchmark (Listrank, see Chapter 6), the program kept terminating with an error. Debugging led to the DMA List Get command. Every time the program tried to get a list, it terminates. This was due to a bug in CellSim. After fixing the bug (Section 5.3.2.1) the Listrank program ran, but now it never terminated. There was another bug in CellSim. At first we modified Listrank to work around this bug, but later on we decided to fix the bug so the results are more accurate (Section 5.3.2.2).

5.3.2.1 DMA Get List bug 1

CellSim terminated the Listrank program when a DMA Get List command occurred with the following error:

```
cellsim:  ../../src/library/simulator/MemoryAccess.h:353:
simulator::address_t simulator::MemoryAccess::get_target() const:
Assertion '_busy' failed.
```

Assertion _busy failed means that a MemmoryAccess object is accessed after it is destroyed, which is not possible. Inspecting the code, the destroying of the Memmory-Acces object is done at line 136 of the file DMAListCommand.hxx in the function processListElement(MemoryAccess * ma):

```
136 MemoryAccess::destroy(ma);
```

Commenting out this line solved the problem.

5.3.2.2 DMA Get List bug 2

After fixing the first bug, the Listrank program ran fine, but never stopped. It was caught in an infinite loop. Debugging showed that the program did not receive data on the right place when using a DMA GETL command. The DMA GETL command fetched data from a main memory address EAL (Effective Address Long) and placed it in the local store at address LSA (Local Store Address). All data was placed in the local store in the first element of a 4-element array with address LSA, while the program expects it in element EAL modulo 4.

At first we corrected this by adjusting the Listrank program to get the value from the correct place. We just used a 0 instead of a modulo 4 operation for the index. Later on, when validating the CellSim results, the number of performance instructions was significantly lower than with the original implementation. Thus we started repairing the bug in CellSim.

Further investigation showed that all values aligned on a 16 byte boundary were correctly placed in the local store, but the others were not. Page 531 of the Cell BE Programming Handbook [16] contains the following sentence:

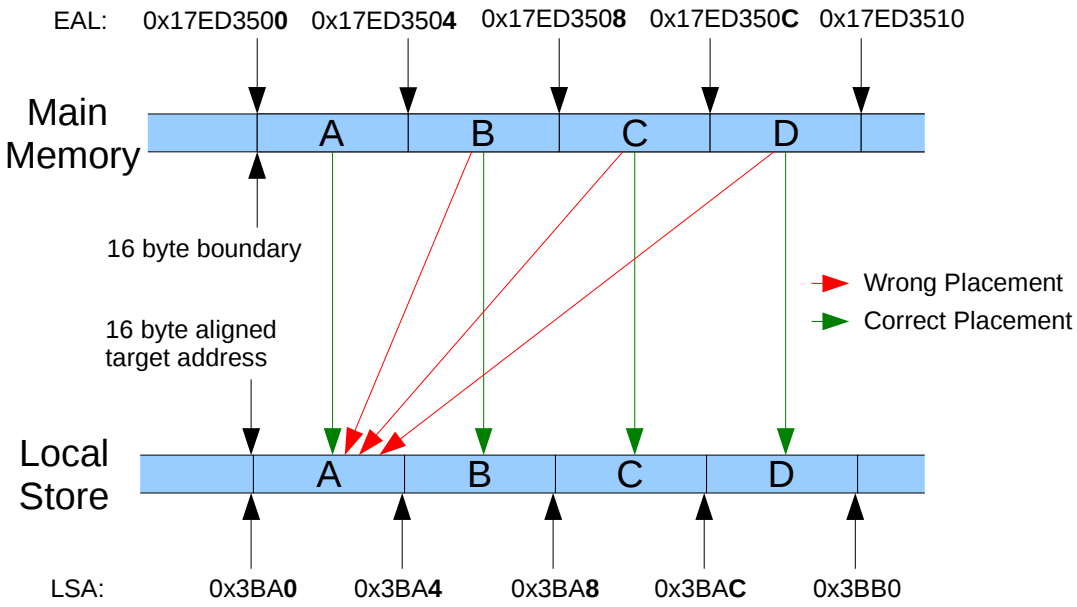

Figure 5.2: DMA Get List command bug in CellSim: Dataparts A to D from Main Memory are all placed in Local Store at A instead of A to D.

> "For list elements smaller than 16 bytes, the least-significant four bits of the transfer-element LSA are equivalent to the least-significant four bits of its EAL."

Thus the 4 bytes of data requested from main memory by the DMA Get List command should be placed in the Local Store in such way that the last 4 bits of the LSA are the same as the requested data's EAL. The target address given to the DMA GETL command should be 16 byte aligned (and is the same for all requests). This is illustrated in Figure 5.2. Data elements A to D are placed in the local store as shown, although the given target address is the same for all.

To make CellSim handle these list requests correctly, we modified the function processListElement(MemoryAccess * ma) in DMAListCommand.hxx. Now, when a transfer of less than 16 bytes is detected, the EALs offset to the 16 byte boundary is calculated and added to the given LSA, which fixes the problem.

5.3.3 Adjusting the Branch Miss Penalty

A branch miss was not simulated correctly in CellSim. The penalty after a branch miss (this occurs when a 'taken' branch is predicted 'not taken', or the other way around) was much lower than it should be. This was corrected before extending CellSim with other branch predictors.

CellSim simulates a branch miss by flushing the instruction buffer. In the next cycle, new instructions are fetched. In the cycle after those instructions arrive in the buffer, the execution continues. The number of cycles needed to fetch an instruction from the local store is defined by de CellSim configuration parameter LS_LATENCY, which is 6 by default. The branch miss penalty then becomes 6 + 1 = 7 cycles. This is much lower than the 18 cycles of the real Cell processor. This is because the pipeline is not simulated.

To get the branch miss penalty closer to reality, we added a counter. This counter (which initial value can be set using BRANCH_MISS_PENALTY in spu.sim) prevents the execution of instructions after they are fetched to the instruction buffer for the specified number of cycles. BRANCH_MISS_PENALTY=11 makes the total branch miss penalty 7 + 11 = 18 cycles, which is the same as for the real SPU. The results of a simulation of ListrankMS using these modification shows that the number of branch miss stall cycles is now differs only 2.1% from the SystemSim results (see Table 6.5).

5.3.4 Extending CellSim with Hint for Branch Instructions

Hint for branch instructions are a static way of branch prediction used by the SPU as described in Section 4.1. However, CellSim did not support hint for branch instructions. The simulator accepted the instructions, but did not execute them. This means that no branches were predicted taken, and thus there were a lot more branch misses in CellSim than on the original Cell SPU. To have a good reference point for further branch prediction extensions in CellSim, it was necessary to add support for hint instructions to CellSim. This section describes the implementation.

The first step in implementing branch hints in CellSim was getting the right information out of the instructions. There are three branch hint instructions [21] of which the binary instruction formats are depicted in Figure 5.3 :

hbr s11, ra The RO field, formed by concatenating the ROH and ROL fields, contains the signed word offset from this instruction to the Branch Instruction Address (BIA). After appending 00 on the right to create the byte offset, the value is sign extended to 11-bit signed value s11. Word element 0 of register ra contains Branch Target Address (BTA). There is also a P-bit. When this is set, the SPU performs an inline prefetch instead of a hinting a branch. This can be used if a program has many loads and stores, which prevent instruction prefetching. Executing this instruction prevents a load/store (because they use the same execution pipeline), so a prefetch can be done. This behavior is not guaranteed in CellSim, because then a load/store instruction and the hbrp instruction can be executed in the same cycle.

hbra s11, s18 The RO field contains the signed word offset from this instruction to the BIA. After appending 00 on the right to create the byte offset, the value is sign extended to 11-bit signed value s11. The I16 field specifies the Branch Target word address. Appending 00 to the right creates the 18-bit BTA (s18)

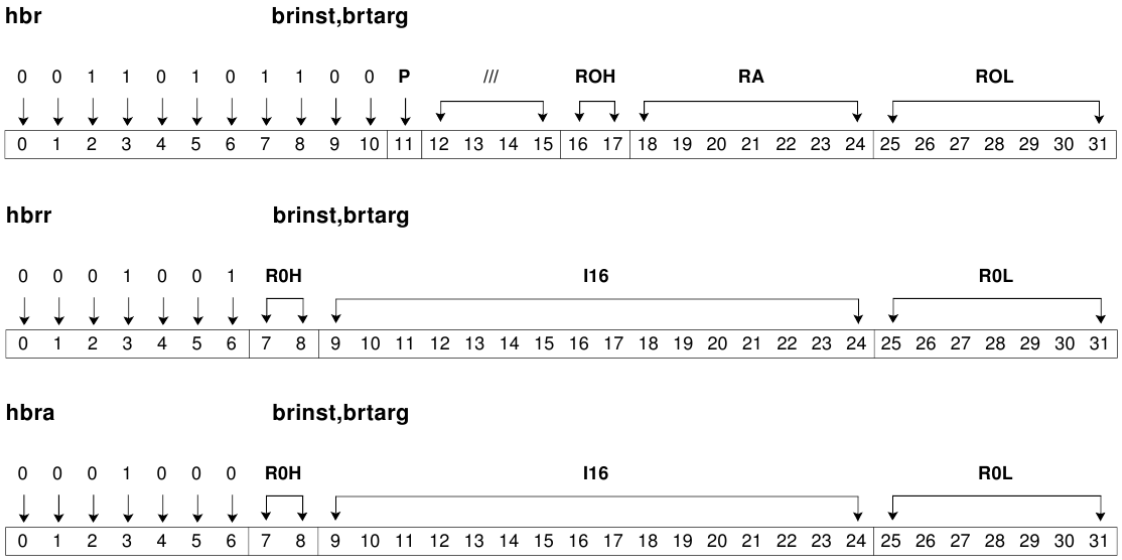

Figure 5.3: Instruction formats of the 3 Hint Instructions, taken from [21].

hbrr s11, s18 The RO field contains the signed word offset from this instruction to the BIA. After appending 00 on the right to create the byte offset, the value is sign extended to 11-bit signed value s11. The I16 field specifies the signed word offset to the Branch Target Address. The BTA is calculated by appending 00 to the right and then sign extend to the 18-bit signed value s18. Adding s18 to the current instruction address gives the BTA.

To distinguish the hint instructions the class Instruction has been modified. The function isBranchHint() was added, which reads the newly added protected boolean _branch_hint. The protected variable _ROH_value and _ROL_value are added to support the RO (= ROH || ROL) field in the branch hint instructions.

Also, a new class BranchHintInstruction is added, which extends the Instruction class. This class contains functions and values specific to Branch Hints:

getBT()	returns Branch Target Address
getBIA()	returns Branch Instruction Addres
getP()	returns value of the P-bit

A hint instruction loads the SPU's Branch Target Buffer (BTB) with the BTA and the BIA. The BTB monitors the instruction stream going into the issue stage of the pipeline. When the instruction address matches the BIA, the SPU continues with the instructions from the BTA. The BTB is defined in the new class BranchTargetBuffer. It also contains control functions regarding hints.

When a hint instruction is executed and its branch target address is more than MIN_BRANCH_DIST instructions away, CellSim stores the hint information in the BTB

using set().

In the next cycle, the instructions from the BTA are prefetched, if there is no instruction fetch in progress. Else this instruction fetch has to finish before the branch target instructions can be fetched.

The prefetched instructions are stored in the BTB instead of the instruction buffer. In this way no modifications are needed to the instruction buffer. In the real SPU, these instructions are placed in a special line of the ILB.

When a branch instruction is executed, the BTB is checked if there is a branch hint available for this branch instruction using the PC. When a hint is available there are two options:

1. The hint is ready for use, which means that the instructions at the BTA are fetched from the local store and stored in the BTB. The instruction buffer is flushed and the prefetched instructions are moved into it. The next cycle, the instruction at the BTA is available for execution. In case of a hint miss (branch not taken), only the instruction buffer is flushed and the instructions need to be fetched from local store.
2. The hint is not ready yet. SPU goes to hint stall mode: the branch is stalled until the BTB gets ready. Then the hint is used as described above.

The hint stays available in the BTB until a new hint instruction is executed or a SYNC or STOP instruction is executed. So the same hint can be used multiple times for the same branch (for example in a loop, the end of loop branch can be hinted outside the loop)

5.4 Summary

In this chapter we described the 2 simulators used. First we discussed IBM Full-System Simulator for the Cell Broadband Processor Version 2.1 (SystemSim), provided with CellSDK 2.1. It simulates the Cell BE cycle accurate, but we cannot modify it.

To implement the proposed branch predictors CellSim is used. This is a modular simulator based on UniSim and it is an instruction set simulator, which means it does not simulate cycle accurate. However, there are a lot of configuration parameters which can be modified in order to get the performance close to reality (which is in our case SystemSim).

Besides extending CellSim with the branch predictors, we also had to make some fixes. First there were 2 bugs regarding DMA List Requests. Second, CellSim does not have support for hint instructions, it simply ignores them. To have a good reference for our branch predictors, hint support was added.

6 Benchmarks

In this chapter the programs used to test the different branch predictors are described. Also the validation of CellSim and the settings used for each benchmark are described. For the first benchmark Listrank, the entire validation process is described to explain how all parameters affect the performance statistics (Section 6.1.3). For the other benchmarks, only a brief explanation of the problems encountered is given, together with the final CellSim settings and validated performance.

We selected these benchmarks using the following constrains: 1. the benchmark can be ported to CellSim, 2. a simulation on CellSim can be done within a reasonable time, and 3. the benchmark needs to have a significant amount of branch miss stall cycles.

6.1 Listrank

The list ranking problem is a fundamental problem for many combinatorial and graph-theoretic applications. It is characterized by fine-grain memory accesses with low locality. The nodes of a linked list are stored in a contiguous memory area. The list ranking problem determines the distance of each node to the head of the list. Bader et al. [22] developed an efficient implementation for the Cell BE. To hide the memory latency, the entire list is divided in sublists. Each SPU processes eight sublists simultaneously in a round robin fashion. While waiting for the DMA request for a node of one list, the other sublists are processed.

The algorithm can be divided into four parts. First, the head of the list is calculated, which is mostly done on the SPU. Second, the list is divided by the PPU into eight sublist per SPU. Third, each SPU calculates the rank of each node to the head of the sublist it is in. Finally, the PPU combines the results and calculates the global rank for each node.

We created two benchmarks out of the Listrank program: First, we use the entire SPU program as a benchmark, and second, we use only part 3 as a benchmark. We did this because part 3 is the only part that has a significant amount of branch miss stall cycles. However, we also wanted to know the effect of our branch predictors on the entire SPU program.

6.1.1 Optimization of Listrank

We selected the Listrank program as a test case for improving the SPU performance. It was made for high performance on the Cell using the SPUs, so it can be used as a benchmark.

The original Listrank program consist of one PPU-program and two SPU-programs. The first SPU-program is for calculating the head of the list, the second is for process-

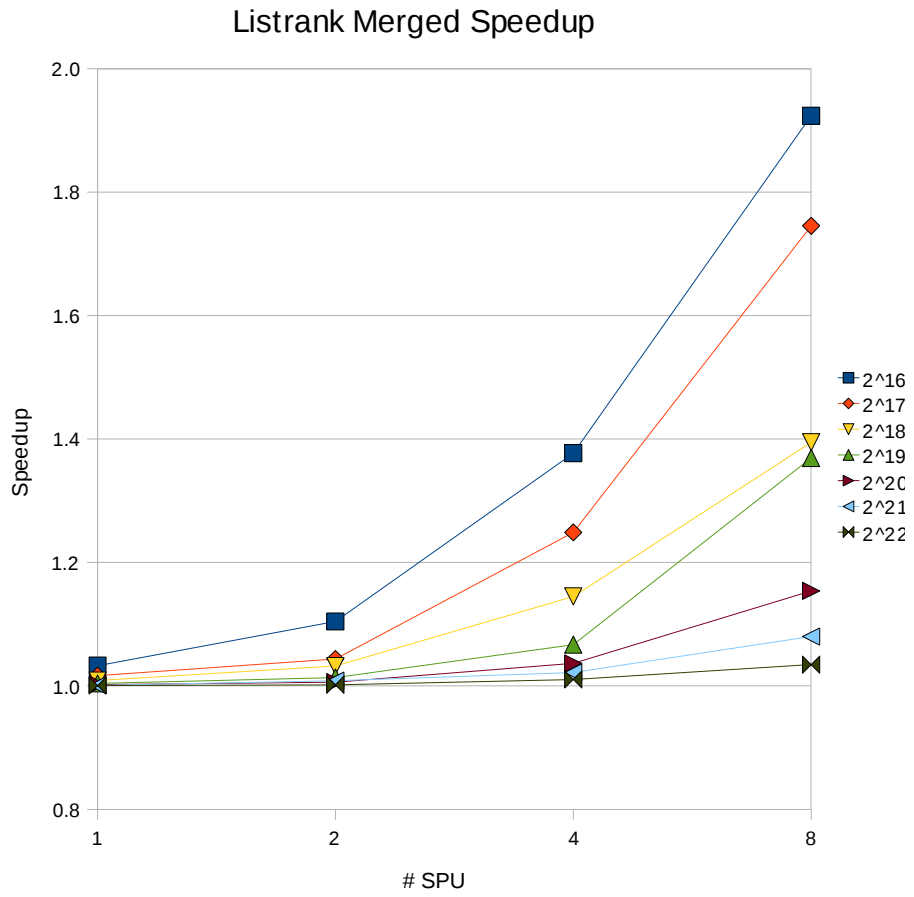

Figure 6.1: Speedup of Listrank with merged SPU programs for ordered lists with different sizes, compared to the original Listrank with two SPU programs.

ing/ranking the eight sublists. This gives unnecessary overhead on the PPU, because for each SPU used, there is an extra creation and destruction of an SPU thread for the second SPU program. Because this overhead depends on the number of SPUs, the listrank program scales worse to large number of SPUs. By merging the two programs into one by using mailboxes as synchronization, this overhead can be eliminated. The mailbox message contains the address of the context block that was normally sent on the start of the SPU program.

With the two SPU programs merged into one, the PPU waits for all SPUs to finish stage 1, using a while loop that tries to read the mailboxes. Another option was to implement this using an interrupting mailbox. Then the PPU suspends until it is interrupted by a mail message from an SPU. The PPU is then available for other tasks. However, CellSim does not support this. There is no significant difference in performance between the two implementations.

The speedup of this modified implementation with respect to the original Listrank

program with two SPU programs is shown in Figure 6.1. For list size 2^{16} and eight SPUs the speedup is almost 2, but for list size 2^{22} it is only 3.5%. The speedup increases with the number of SPUs. The time saved by not creating and destroying a thread for the second SPU program depends on the number of SPUs used. For larger list sizes, the speedup is less, because then time needed for thread creation and destruction is a smaller portion of the total execution time.

The Listrank program was also modified to accept the command line parameter #spus, which gives the number of SPUs to use. This version is called Listrank Merge Scalable (ListrankMS) and is used as benchmark. The benchmark that only uses part 3 of the algorithm is called ListrankMS-P3.

ListrankMS can rank two classes of list, ordered and random. For an ordered list, all elements are placed on positions corresponding with their rank. Thus the i^{th} element in the list has rank i and its successor is element *i+1*. For a random list, successive elements are placed randomly in the array. The branching behavior of ListrankMS is the same for both lists. Therefore we use a ordered list. To keep the simulation time low, we use a list with size 2^{16}.

6.1.2 Analysis of ListrankMS

IBM SystemSim provides different ways to analyze the performance of an application [18]. We used the SPU performance statistics for analyzing the performance of ListrankMS. To obtain the statistics, the simulation mode was set to 'cycle'. Also commands to start, stop, and clear the profiling were added to the SPU code in order to profile only the part of the code which is doing the calculations. In this way we can leave out the cycles needed for initializing and terminating the SPU, which is not accurately modeled by CellSim, and thus gives a more accurate comparison.

The results are depicted in Figure 6.2. The chart shows how each of the performance statistics described in Table 6.1 contributes to the total number of performance cycles. The number of performance cycles is the part of the total execution cycles that is profiled using the profiling commands. The results are quite similar for random and sequential lists, but between different list sizes there is a large difference. This difference is mainly due to the smaller fraction of channel stall cycles for large lists. Most of these channel stall cycles occur when the SPU waits for the PPU to divide the sublists. The amount of calculations for this does not scale with the list size, but with the number of sublists and thus is almost constant.

A large portion of stall cycles is due to dependencies. 20% for a list with 2^{16} elements and 45.8% for size 2^{22}. Because the code to calculate the head (summation) and the rank is simple and consist of few statements, little can be improved here.

Also, there are a significant amount of branch stall cycles. These are stalls after a branch miss. The processing of 8 sublists has led to a code with many if-statements. Besides that, there are some loops and thus there are a lot of branches. The hint instructions used to hint branches work fine for the loops, but not for the if-statements. Ranking a 2^{16} elements sequential list needs 333K branches. 140K hints instructions are used, which gave 131K hits. This seems quite good, but one hint instruction can lead to multiple hint hits. In this case, about half of the hint hits are due to 31 hint

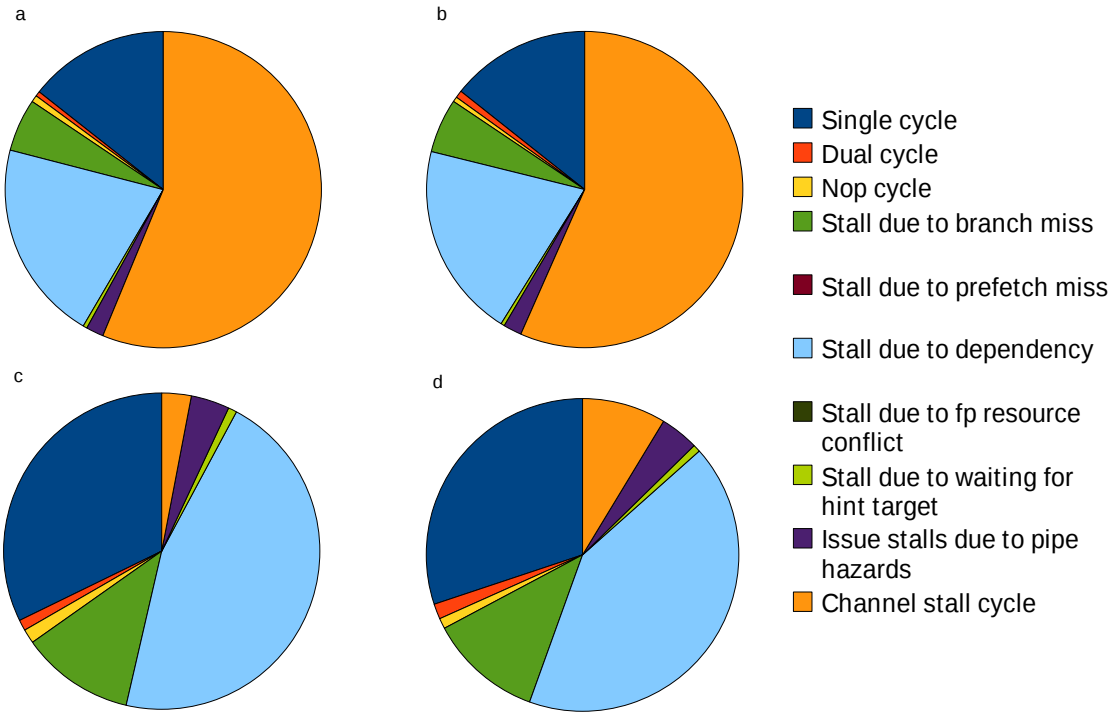

Figure 6.2: Breakdown of the performance cycles for ListrankMS: a) Random list size 2^{16}, b) Sequential list size 2^{16}, c) Random list size 2^{22}, d) Sequential list size 2^{22}.

Table 6.1: SPU Statistics Explained.

Single cycle	Cycles in which only 1 non-NOP instruction was executed
Dual cycle	Cycles in which 2 non-NOP instructions were executed
NOP cycle	Cycles in which only NOP instructions were executed
Stall due to branch miss	Cycles in which branch mispredict prevented any instruction from executing
Stall due to prefetch miss	Cycles in which instruction run-out occurred
Stall due to dependency	Cycles in which source/target operand dependencies prevented any instruction from being issued
Stall due to fp resource conflict	Cycles in which shared use of FPU stages prevented any instruction from being issued (e.g. FXB, FP6, FP7, FPD)
Stall due to waiting for hint target	Cycles for which target load delay for a hinted branch prevented instruction fetch
Issue stalls due to pipe hazards	Cycles for which pipeline scheduling hazards prevented instruction issue
Channel stall cycle	Cycles for which the pipeline was stalled waiting on channel operations to complete

Table 6.2: SPU Statistics for SystemSim and CellSim with original settings for ListrankMS with a sequential list of size 2^{16}. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	44856696	100.0%	18092508	100.0%	-26764188	-59.6%
fetch_stall_cycles	0	0.0%	66151	0.4%	66151	∞%
dependency_stall_cycles	8353631	18.6%	9289640	50.8%	936009	11.2%
channel_blocked_cycles	26738143	59.6%	2873	0.0%	-26735270	-100.0%
branch_miss_stall_cycles	2318048	5.2%	2458302	13.4%	140254	6.1%
	Instr.		Instr.		Instr.	
performance_instructions	7044541		7044557		16	0.0%
branch_instructions	333001		333001		0	0.0%
hint_instructions	140072		140072		0	0.0%
hint_hits	131844		131844		0	0.0%

instructions. This means that about half of the hints do not lead to a hint hit. A wrong hint instruction leads to a branch miss stall penalty of 18 cycles. The branch predictors we proposed in Chapter 4 can handle this type of branches much better and could give a speedup.

6.1.3 Performance Validation: IBM SystemSim versus CellSim

Table 6.2 shows the first results of ListrankMS on CellSim compared to SystemSim. Besides a selection of the most important performance statistics mentioned earlier, the number of performance cycles and instructions are given. Also the number of executed branch and hint instructions are given, together with the number of hint hits (number of correctly hinted branches).

The results obtained with CellSim are very different from those obtained with IBM SystemSim, even with the same compiler settings (spu-gcc 4.1.1 with options -O3 -W -Wall -Winline -Wno-main). To get useful results from CellSim, this had to be solved. For better understanding the differences, the SPU code was split into three parts using the profiler clear, start and stop functions: 1) Calculating the head node, 2) wait for sublists, and 3) calculating sublist ranks. ListrankMS was executed with listsize 16, listtype 1 (sequential) and one SPU.

6.1.3.1 Dependency Stalls

The number of stalls due to dependencies was 11% larger for CellSim than for SystemSim. Investigation of the CellSim settings showed that all the pipeline function unit latencies were set to 6 cycles. Using the correct values (as found on page 690-1 of [16]) gives much better results. The Latency for Load/Store instructions (LS) is set to 0, because this is already modeled in the Local Store latency (LS_LATENCY). Also, the Branch

Table 6.3: Latency of SPU Pipeline Functional Units.

Instruction Type	**SPU**	**CellSim**
FP6	6	6
FP7	7	7
FX2	2	2
PPD	7	7
FX3	4	4
FXB	4	4
NOP	0	0
SHUF	4	4
LS	6	0
BR	4	0
SPR	6	6
LNOP	0	0

latency is set to 0. CellSim executes them right away, so this number does not affect the simulation. The full list of used latencies compared with the real SPU values is given in Table 6.3

Getting the correct number of dependency stall cycles is very difficult, because the SPU pipeline is not simulated fully. Therefore CellSim has different rules to determine if there can be a dual issue, single issue or no issue at all. For example, instructions that use the same pipeline can be executed in one cycle in CellSim, which is not possible on the real SPU. If this happens between two instructions with a data dependency, the number of dependency stall cycles can be different. The number of instructions executed in one cycle is determined by the SPU Issue Width. This parameter is changed in Section 6.1.3.4.

6.1.3.2 Issue Stalls due to Pipe Hazards

SystemSim results in Table 6.2 show 786432 issue stall cycles due to pipe hazards. These happen when the pipeline is not ready to receive the next instruction. CellSim does not measure it, because this behavior is not modeled at all (due to the pipeline stages that are not modeled).

The issue stalls are generated by two double-precision floating point instructions which are part of the loop that sums the list elements in order to calculate the start of the list. These are the instructions **dfs** and **dfa** in the loop code depicted in Listing 6.1.

The two double-precision floating point instructions are right after each other. In the SPU no instructions can be issued in the six cycles after a double-precision floating point instruction is issued. So in each loop, the pipeline is stalled twice for six cycles. Having 2^{16} elements to add, the total number of stall cycles is 786432 which corresponds with the number stall cycles in SystemSim.

Listing 6.1: Loop code in ListrankMS with double floating points instructions.

```
5e0:  18020305    a        $5,$6,$8
5e4:  38820303    lqx      $3,$6,$8
5e8:  41400002    ilhu     $2,32768
5ec:  40820f15    il       $21,1054
5f0:  1c010306    ai       $6,$6,4
5f4:  78024304    ceq      $4,$6,$9
5f8:  3b814183    rotqby   $3,$3,$5
5fc:  48208182    xor      $2,$3,$2
600:  54a00103    clz      $3,$2
604:  b60c105     shl      $5,$2,$3
608:  7c080194    ceqi     $20,$3,32
60c:  8054195     sf       $21,$3,$21
610:  18014285    a        $5,$5,$5
614:  58250a95    andc     $21,$21,$20
618:  b2a14a8a    shufb    $21,$21,$5,$10
61c:  3f610a82    shlqbii  $2,$21,4
620:  59a2c102    dfs      $2,$2,$11
624:  59808387    dfa      $7,$7,$2
628:  4020007f    nop      $127
62c:  207ff684    brz      $4,0x3ffb4
```

However, part of those cycles can be found in the CellSim statistics. The two double-precision floating point instructions have a data dependency. In SystemSim, six of the seven cycles of this dependency are hidden by the issue stalls. But not for CellSim, there those stall cycles are just dependency stall cycles. Which means CellSim has more dependency stall cycles than SystemSim, namely half of the issue stall cycles, which are 393216 cycles. The other half is not simulated and thus the number of performance cycles is lowered by the same amount.

6.1.3.3 Fetch Stalls

The adjusted pipeline latency settings have introduced new fetch stall cycles in CellSim. Investigation of the SPU trace shows that the pipeline runs out of instructions, especially when a hint target is fetched. This is because the simulated instruction buffer size (16 instructions) and fetch width (8 instructions) are much lower than for the real SPU (76 and 32 instructions respectively) to increase simulation speed. Increasing the instruction buffer size to 32 and the fetch width to 16 instructions, makes the number of fetch stall cycles zero.

But it also leads to slightly more dependency stall cycles, because some of those were hidden by fetch stall cycles. Besides that, the number of branch miss stall cycles is decreased. After a branch, the new target instructions are fetched. When the SPU is busy fetching other (now unnecessary) instructions, it has to wait till it is finished before a new fetch can be done. These cycles are counted as branch miss stall cycles. If there are less instruction fetches because of the larger buffer size and fetch width, this happens less.

6.1.3.4 Performance Cycles

The number of performance cycles in Table 6.2 differs. CellSim always tries to issue two instructions every cycle, even if they use the same pipeline. This is not the correct behavior. But in the file default_configuration_parameters.h the parameter SPU issue width can be changed. When the issue width is set to 1.5, CellSim issues on average 1.5 instructions per cycle. This means that the first cycle it issues one instruction, the following two instructions and the next one instruction again.

The SPU issue width does not only affect the number of performance cycles, but it also has affect on the number of dependency stall cycles and fetch stall cycles. Increasing the issue width gives more dependency stall cycles. Also the number of fetch stall cycles could increase, because the SPU runs out of instructions earlier.

Changing the SPU issue width to 1.8 makes the performance cycle count close to that of SystemSim, especially for part 3. Although it is a little larger, the percentages of dependency stall and branch miss stalls are almost identical for both CellSim and SystemSim.

6.1.3.5 Channel Stalls

The SystemSim statistics in Table 6.2 show that 59.6% of the performance cycles are Channel Stalls. These happen when the SPU has to wait for a channel operation to complete. These are operations for communication between the SPU and memory, PPU or another SPU like DMA transfers and Mailbox Read and Writes.

CellSim has only 0.01% Channel Stalls. When different parts are investigated, most of the stalls occur in part 2 for SystemSim. CellSim has no channel stalls for stage 2.

In part 2, the SPU sends a mail to the PPU and then waits until it gets a mail back. Inspecting the event log of SystemSim shows that most channel stall cycles are due to waiting on the mail to return. When the SystemSim simulation mode is changed to Simple (instead of Cycle) and the SPU mode is set to Pipe, the simulator still does a cycle accurate simulation of the SPU, but not anymore for the PPU and the memory. Now the number of Channel Stalls is reduced to only 0.26% and thus the difference with CellSim is much less.

Also, in part 3 SystemSim has 245K channel stall cycles more than CellSim. The SystemSim event log shows that channel stalls happen when the SPU writes the partial results back to memory, after the for loop. The SystemSim event log shows that these stalls are due to a memory address translation fault. These do not occur in CellSim so there are no stalls for that.

6.1.3.6 Final Settings

Table 6.4 shows the final CellSim settings we will use for the ListrankMS benchmarks. With those settings, the performance statistics of CellSim are now quite similar to those of SystemSim with simulation mode 'simple' and SPU mode 'pipe'. In Table 6.5 statistics for the ListrankMS SPU program are depicted.

But part 3 of the ListrankMS program is the most interesting part, because this is the part with the most stall cycles due to branch misses. The statistics for this part are

Table 6.4: CellSim Configuration for ListrankMS.

Parameter	**Value**
BRANCH_MISS_PENALTY	11
HINT_TRIGGER_DELAY	8
IBUFF_CAP	32
FETCH_WIDTH	16
SPU Issue Width	1.8
LDSTQ_SIZE	1

Table 6.5: SPU Statistics for SystemSim and CellSim with adjusted settings for ListrankMS program with a sequential list of size 2^{16}. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	18437301	100.0%	18288224	100.0%	-14907	-0.8%
fetch_stall_cycles	0	0.0%	0	0.0%	0	0.0%
dependency_stall_cycles	8353631	45.3%	8977009	49.1%	623378	7.5%
channel_blocked_cycles	318750	1.7%	2872	0.0%	-315878	-99.1%
branch_miss_stall_cycles	2318048	12.6%	2367438	13.0%	49390	2.1%
	Instr.		Instr.		Instr.	
performance_instructions	7044539		7044557		18	0.0%
branch_instructions	333001		333001		0	0.0%
hint_instructions	140072		140072		0	0.0%
hint_hits	131844		131844		0	0.0%

Table 6.6: SPU Statistics for SystemSim and CellSim with adjusted settings for ListrankMS-P3 with a sequential list of size 2^{16}. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	15085386	100.0%	15662336	100.0%	576950	3.8%
fetch_stall_cycles	0	0%	0	0%	0	0.0%
dependency_stall_cycles	7042185	46.7%	7464861	47.7%	422676	6.0%
channel_blocked_cycles	245420	1.6%	268	0%	-245152	-99.9%
branch_miss_stall_cycles	2317758	15.4%	2367132	15.1%	49374	2.1%
	Instr.		Instr.		Instr.	
performance_instructions	5733021		5733051		30	0.0%
branch_instructions	267449		267449		0	0.0%
hint_instructions	140041		140041		0	0.0%
hint_hits	66310		66310		0	0.0%

in Table 6.6. Although CellSim uses more cycles, the percentage of branch miss stall cycles and dependency stall cycles are almost identical for CellSim and SystemSim.

6.2 MergeSort

MergeSort is an SPU-only implementation of the recursive merge-sort algorithm. An array of elements is sorted by dividing it into two equally sized arrays, which are then sorted using merge-sort. When they are sorted, the two arrays are merged into one array containing the sorted elements.

The input array is created by the SPU. For testing, 1024 elements are used. It is a decreasing sequence starting at 1024 and ending with 1. Because the branching behavior depends on the input, we also implemented a random input array of equal size. This benchmark is referred to as MergeSort Random.

Table 6.7: CellSim Configuration for MergeSort.

Parameter	**Value**
BRANCH_MISS_PENALTY	11
HINT_TRIGGER_DELAY	8
IBUFF_CAP	32
FETCH_WIDTH	16
SPU Issue Width	1.7
LDSTQ_SIZE	1

Table 6.8: SPU Statistics for MergeSort at IBM SystemSim compared to CellSim. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	672244	100.0%	685548	100.0%	13304	2.0%
fetch_stall_cycles	0	0%	7	0%	7	∞%
dependency_stall_cycles	220546	32.8%	214717	31.2%	0	0.0%
channel_blocked_cycles	0	0%	0	0%	-5829	-2.6%
branch_miss_stall_cycles	147595	22.0%	148709	21.7%	1114	0.8%
	Instr.		Instr.		Instr.	
performance_instructions	342176		342175		-1	0.0%
branch_instructions	37000		37000		0	0.0%
hint_instructions	7195		7195		0	0.0%
hint_hits	20525		20525		0	0.0%

Table 6.9: SPU Statistics for MergeSort Random at IBM SystemSim compared to CellSim. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	712395	100.0%	744114	100.0%	31719	4.5%
fetch_stall_cycles	0	0%	7	0%	7	∞%
dependency_stall_cycles	237402	33.3%	244623	32.9%	7221	3.0%
channel_blocked_cycles	0	0%	0	0%	0	0.0%
branch_miss_stall_cycles	131679	18.5%	138545	18.6%	6866	5.2%
	Instr.		Instr.		Instr.	
performance_instructions	381514		381515		1	0.0%
branch_instructions	39179		39179		0	0.0%
hint_instructions	11023		11023		0	0.0%
hint_hits	20018		20018		0	0.0%

6.2.1 Performance Validation

To validate the results on CellSim, the MergeSort programs were also profiled with IBM SystemSim. These results were used as reference to fine tune the different CellSim parameters. Finally, the parameters as shown in Table 6.7 make the CellSim results approach IBM SystemSim results the closest for both the ordered and random input. Table 6.8 and 6.9 show the performance statistics of CellSim compared to IBM SystemSim.

6.3 QuickSort

QuickSort is a SPU-only implementation of the recursive quick-sort algorithm. First, an array element in the middle is chosen as so-called pivot. Then, the array is divided into two parts. Elements with a value smaller than the pivot value are placed in the lower part and larger values in the higher part. When the array is divided, both lower and higher part are sorted using a recursive call to quick-sort. The recursion stops when the array contains one or zero elements (thus is sorted).

As was the case in MergeSort, the input array is created by the SPU. For testing, 1024 elements are used. It is a decreasing sequence starting at 1024 and ending with 1. Because the branching behavior is depends on the input, we also implemented a random input array of equal size. This benchmark is referred to as QuickSort Random.

6.3.1 Performance Validation

To validate the results on CellSim, the QuickSort programs were also profiled with IBM SystemSim. These results were used as reference to fine tune the different CellSim parameters. Finally, the parameters as shown in Table 6.10 make the CellSim results approach IBM SystemSim results the closest for both the ordered and random input. Tables 6.11 and 6.12 show the performance statistics of CellSim compared to IBM SystemSim.

Table 6.10: CellSim Configuration for QuickSort.

Parameter	**Value**
BRANCH_MISS_PENALTY	11
HINT_TRIGGER_DELAY	8
IBUFF_CAP	32
FETCH_WIDTH	16
SPU Issue Width	1.7
LDSTQ_SIZE	1

Table 6.11: SPU Statistics for QuickSort at IBM SystemSim compared to CellSim. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	363944	100.0%	377308	100.0%	13364	3.7%
fetch_stall_cycles	0	0%	0	0%	0	0.0%
dependency_stall_cycles	88706	24.4%	93882	24.8%	5176	5.8%
channel_blocked_cycles	0	0%	0	0%	0	0.0%
branch_miss_stall_cycles	167038	45.9%	171288	45.4%	4250	2.5%
	Instr.		Instr.		Instr.	
performance_instructions	114491		114495		4	0.0%
branch_instructions	16285		16284		-1	0.0%
hint_instructions	1537		1538		1	0.1%
hint_hits	3604		3605		1	0.0%

Table 6.12: SPU Statistics for QuickSort Random at IBM SystemSim compared to CellSim. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	538141	100.0%	551716	100.0%	13575	2.5%
fetch_stall_cycles	0	0%	0	0%	0	0.0%
dependency_stall_cycles	146019	27.1%	153141	27.8%	7122	4.9%
channel_blocked_cycles	0	0%	0	0%	0	0.0%
branch_miss_stall_cycles	212671	39.5%	218764	39.7%	6093	2.9%
	Instr.		Instr.		Instr.	
performance_instructions	182640		182644		4	0.0%
branch_instructions	24426		24425		-1	0.0%
hint_instructions	4126		4127		1	0.0%
hint_hits	4277		4278		1	0.0%

Table 6.13: SPU Statistics for ClustalW at IBM SystemSim. The percentages are the number of stall cycles as percentage of performance cycles.

Counter	**SystemSim**	
	Cycles	%
performance_cycles	104746645	100.0%
fetch_stall_cycles	0	0%
dependency_stall_cycles	43281987	41.3%
channel_blocked_cycles	0	0%
branch_miss_stall_cycles	35566	0%
	Instructions	
performance_instructions	72603137	
branch_instructions	1398878	
hint_instructions	1019	
hint_hits	1395828	

6.4 ClustalW

ClustalW is a sequence alignment application used in the field of bioinformatics to compare multiple biological sequences like DNA, proteins and RNA. It uses a slightly modified Needleman-Wunsch algorithm to study the similarity between one sequence and many others. We use an SPU implementation made by Isaza et al [23]. The algorithm consists of 3 steps: 1) All-to-all pair wise alignment, 2) Creation of a phylogenetic tree and 3) Use of the phylogenetic tree to carry out a multiple alignment. Profiling showed that 70% of total execution time was used for step 1 by the function forward_pass. This function calculates the similarity of two sequences. It was implemented on the SPU and is used as a benchmark. Two sequences, containing about 2000 symbols each, are used as input.

We profiled the ClustalW kernel using SystemSim. As the results in Table 6.13 show, there are few branch miss stall cycles. There are a lot of branches, but their behavior is very predictable. Most of these branches are correctly hinted using only 1019 hints. This means that there is almost no speedup possible by using a branch predictor and thus it is not an interesting benchmark.

6.5 MiniGZip

MiniGZip is an SPU implementation of the GZIP (de)compression program based on the ZLIB library. The MiniGZip program uses the SPU optimized version of ZLIB Library (version 2.0) implemented by Seunghwa Kang [24]. Various optimization techniques like SIMDization, branch hints and loop unrolling are used. The code is also parallelized, thus it can use multiple SPUs. Load balancing is achieved using a work queue.

As input for the benchmark, a 39 KB JPEG image is used. MiniGZip is run without options, which means it compresses the image on one SPU. It uses both LZ77 and Huffman coding and the compression level is 6. The program and the library are compiled

using gcc with the standard makefiles which are provided in the package.

6.5.1 Porting to CellSim

In order to use the MiniGZip program on CellSim, some modifications were needed to the PPU part of the program. Out of the box a lot of errors occurred. First, all functions related to semaphore.h were removed, because they are not supported by CellSim. Also, a copy of free_align.h was added to the include directory of CellSim. This was not there before, but is needed by the program. Now the program can compile, but while running CellSim reports systemcalls that are not implemented. These are the functions *sched_yield()* and *usleep()*, which function calls we removed from the code. Then there is an fstat error. Fstat is not supported, thus the file_stat.size statements were replaced by a new variable file_size. At last a standard input filename was set, because no command line parameter can be passed to the program by CellSim.

After all those changes, the program runs on CellSim. A lot of synchronization logic is removed, but because only 1 SPU is used in our experiments, this is not a problem. But removing the instructions makes the program incapable of creating a correct output file. However, all these changes are made in de PPU part of the program and the SPU part is unchanged. In the next section can be seen that the number of executed instructions on the SPU is almost identical. Also the program flow is not affected, which can be seen by the number of (taken) branches. This indicates that the SPU program executes correctly and can be used as a benchmark.

6.5.2 Performance Validation

To validate the results on CellSim, the ClustalW kernel was also profiled with IBM SystemSim. Changing the CellSim configuration as described for the other benchmarks, we get the results depicted in Table 6.14. Most of the results are quite similar, but the number of hint hits is not! Running at SystemSim, there are 19% more hint hits than with CellSim. This is strange, because the number of performance, branch and hint instructions is the same on both. Also the number of taken branches is the same. The number of hint hits should also be the same.

6.5.2.1 Solving the Hint Hits Difference

The large difference in hint hits led to a significant higher amount of branch miss stall cycles, because when there is a hint hit, the branch penalty is reduced (sometimes even till 0 cycles). To have a good benchmark for branch prediction, the branch behavior (which includes hints) should be representative. Thus the cause of the difference in hint hits should be found.

First, the part of the code that is responsible for the difference was identified by narrowing the profiling step by step. Most of the difference is caused by the do-while-loop in the function *longest_match()* in deflate.c of the ZLIB library. On IBM SystemSim, this function has 31704 hint instructions that give 15990 hint hits. On CellSim, the same function generates 523 hits. This is the exact difference as found earlier. The function does not contain very special code that is not supported

Table 6.14: SPU Statistics for MiniGZip Compression at IBM SystemSim compared to CellSim with MIN_BRANCH_DIST=8. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	22431156	100.0%	23318383	100.0%	869187	3.9%
fetch_stall_cycles	10	0%	84	0%	74	740.0%
dependency_stall_cycles	9167727	40.9%	9193783	39.4%	26056	0.3%
channel_blocked_cycles	500341	2.2%	12981	0.1%	-505400	97.5%
branch_miss_stall_cycles	2485953	11.1%	2712869	11.6%	226916	9.1%
	Instr.		Instr.		Instr.	
performance_instructions	12425240		12425238		-2	0.0%
branch_instructions	619438		619438		0	0.0%
hint_instructions	58829		58829		0	0.0%
hint_hits	96297		80830		-15467	-16.1%

by CellSim, thus this can not be the cause. Therefore the fault should be in the simulator.

One of the CellSim parameters that can influence the number of hint hits is MIN_BRANCH_DIST. It defines the number of instructions that should be between the hint and its branch, based on the instruction addresses. When this difference is smaller or equal than MIN_BRANCH_DIST, the hint instruction is not used. Normally, it is set to 8. The Cell BE Programming Handbook [16] states that four instruction pairs must separate the hint from the branch in order for the branch to be predicted to the taken path, which means eight instructions. This is because a hint instruction takes four pipeline stages to complete. Only after completion, the hint trigger is set and a branch can trigger it. If a branch enters the execution pipeline before that, the hint is not ready and can not be used.

Table 6.15: Hint instructions which are to close to the branch in MiniGZip. The hint instruction is used count times. Offset is the distance between hint and branch instruction.

Hint Instruction Address	**Count**	**Offset**
0x10764	7	8
0x1038c	8	8
0x3e6c	30366	6
0x3fc4	694	6
0x12b4	1	6
0x12d4	1	6
0xf35c	1	6
0x11d30	1	8

Listing 6.2: MiniGZip output trace with Odd/Even pipeline indication.

```
0x3E6C:  1007c306      hbra     0x18,0x3e18      Odd
0x3E70:  3884c502      lqx      $2,$10,$19       Odd
0x3E74:  3b820102      rotqby   $2,$2,$8         Odd
0x3E78:  1822c10d      and      $13,$2,$11       Even
0x3E7C:  58064683      clgt     $3,$13,$25       Even
0x3E80:  20000103      brz      $3,0x8           Odd
0x3E84:  217ff289      brnz     $9,0x3ff94       Odd
```

When MIN_BRANCH_DIST=0 is used, a lot more hint hits occur: 19173 hits in the function longest_match() and 99426 hits in total (see Table 6.17). So now there are 3129 hits more on CellSim than on IBM SystemSim. Adding some more counters to CellSim to gather some statistics about the hint misses give some interesting results (see Table 6.15). There are eight hint instructions that are eight or less instructions away from their branch. Three of them have seven instructions between the hint and the branch, the other five are separated by five instructions.

The hint at address 0x3E6C is used 30366 times, so could be the cause of problem. The output trace given in Listing 6.2 of CellSim shows the following code (Odd/Even is the pipeline in which the instruction is executed).

Just before the hinted branch is another branch. Based on a comparison, it makes the program skip the hinted branch or not. When it is not skipped, the branch is hinted correctly most of the time.

Nevertheless SystemSim uses this hint. This could be explained as followed. As mentioned earlier, the hint instruction needs four execution pipeline stages to complete. When the hint has left the pipeline, the branch can enter. The five instructions above are executed in five separate stages because of dependencies or usage of the same pipeline as depicted in Listing 6.3.

Although there are less than eight instructions, there are more then four pipeline stages between the hint and the branch. The SPU pipeline will insert NOP and LNOP instructions in case of a single issue. The five instructions are then transformed into five instruction pairs and the hint is ready for usage when the branch enters the execution pipeline.

Listing 6.3: MiniGZip code with dependency indication.

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

CellSim does not simulate the SPU pipeline stage by stage and thus cannot handle this kind of behavior. The number of instructions executed per cycle depends on the parameter SPU Issue Width. No NOP/LNOP instructions are padded into the instruction stream so there is no way to determine the number of stages between a hint and its branch. Setting MIN_BRANCH_DIST=5 gives the same result for this program.

We could not find any reason why now there are 3129 more hint hits on CellSim. The other hint instructions mentioned in Table 6.15 have the same behavior. Also, none of them (including 0x3E6C) are placed outside a loop which could make them hint a branch multiple times.

Finally, the CellSim configuration parameters as shown in Table 6.16 are used for the benchmarks. Table 6.17 shows the profiling statistics when these parameters are used.

Table 6.16: CellSim Configuration for MiniGZip Compression.

Parameter	**Value**
BRANCH_MISS_PENALTY	10
HINT_TRIGGER_DELAY	8
IBUFF_CAP	48
FETCH_WIDTH	24
SPU Issue Width	1.7
LDSTQ_SIZE	1
MIN_BRANCH_DIST	5

Table 6.17: SPU Statistics for MiniGZip Compression at IBM SystemSim compared to CellSim with MIN_BRANCH_DIST=5. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	22431156	100.0%	23003099	100.0%	552872	2.5%
fetch_stall_cycles	10	0%	84	0%	74	0.3%
dependency_stall_cycles	9167727	40.9%	9193905	40.0%	26178	0.3%
channel_blocked_cycles	500341	2.2%	12980	0.1%	-505401	-97.5%
branch_miss_stall_cycles	2485953	11.1%	2396457	10.4%	-89496	-3.6%
	Instr.		Instr.		Instr.	
performance_instructions	12425240		12425572		-2	0.0%
branch_instructions	619438		619438		0	0.0%
hint_instructions	58829		58829		0	0.0%
hint_hits	96297		99426		3129	3.2%

6.5.2.2 Compiling with Branch Warnings

When both the library and the SPU Compress program are compiled with the modified compiler used for the branch warnings, CellSim generates the following error when running the program:

```
minigzip:  gzopen failure
```

This error is thrown on line 223 of minigzip_spu_compress.c, after the following statement on line 217 returned NULL:

```
out = gzopen_w( a_outfile, cb.a_outmode, p_out, write_data_dma );
```

Compiling the library with the modified compiler and using the normal compiler for the SPU Compress program solves the problem. The application spends most of its time execution functions from the library, so this only affects the results slightly. We could not find the source of the problem.

6.6 SPE-JPEG

SPE-JPEG is a program made by Vitaly Vidmirov, that decodes a JPEG-image on using the SPU. I 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 program uses 1 SPU to decode the image, using vectorized Huffman decoding, SIMDimized IDCT and colorspace conversion, and double buffering.

The code was originally made to show the image on a PS3. The used simulators only have a text output, so this part of the (PPU) code was removed. In the SPU code, the *mfc_getb()* statements are replaced with *mfc_get()*, because *mfc_getb()* is not supported on CellSim. The difference between the two DMA get request is that with *mfc_get()*, the SPU can change the order in which the DMA request are processed to improve performance, while with *mfc_getb()* the order is preserved. Simulation shows that the performance statistics using *mfc_getb()* and *mfc_get()* are exactly the same, so in this case it does not matter which get command is used.

To compile the program, two modifications were made to the Makefile. First, the program is compiled with option -O3. Originally this option was not used, thus it contains no hints. With -O3 the program has hints and is a lot faster (3K instead of 20K performance cycles, mainly due to a reduction of dependency stall cycles). Second, the option -std=c99 is replaced by -std=gnu99 to support profiling on CellSim. The profiling commands are implemented as assembly instructions, which are not supported by the c99 standard.

6.6.1 Performance Validation

To validate the results on CellSim, the SPE-JPEG program was also profiled with IBM SystemSim. Using the CellSim configuration parameters as shown in Table 6.18 give the

Table 6.18: CellSim Configuration for SPE-JPEG.

Parameter	**Value**
BRANCH_MISS_PENALTY	10
HINT_TRIGGER_DELAY	8
IBUFF_CAP	40
FETCH_WIDTH	20
SPU Issue Width	1.7
LDSTQ_SIZE	1
MIN_BRANCH_DIST	8

Table 6.19: SPU Statistics for SPE-JPEG at IBM SystemSim compared to CellSim. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	3278101	100.0%	3132410	100.0%	-145691	-4.4%
fetch_stall_cycles	3840	0.1%	91	0.0%	-3749	-97.6%
dependency_stall_cycles	576591	17.6%	564805	18.0%	-11786	2.0%
channel_blocked_cycles	41166	1.3%	5352	0.2%	-35814	-87.0%
branch_miss_stall_cycles	476574	14.5%	479459	15.3%	2885	0.6%
	Instr.		Instr.		Instr.	
performance_instructions	2708609		2708609		0	0.0%
branch_instructions	95120		95120		0	0.0%
hint_instructions	20961		20961		0	0.0%
hint_hits	18754		18754		0	0.0%

results in Table 6.19.

SystemSim 3.1 beta is used for this benchmark and the following, because the computer with version 2.1 crashed. This version does not provide the number of hint hits directly. To get this number, request the hint statistics using the command "mysim spu n display statistics hint". At the bottom of these statistics is the number of unhinted instruction sequence errors. The number of hint hits is calculated using this formula:

Hint Hits = Branch taken - Unhinted instruction sequence errors

6.7 Deblocking Filter

The Deblocking Filter is a kernel from the H.264 video processing coder/decoder [25]. The video bitstream created by the H.264 codec is half the size of the bitstream generated using the MPEG-4 standard. To sustain the same quality, the computational complexity is increased. The Deblocking Filter is a way to improve the quality of a video image.

The H.264 codec uses a discrete cosine transform for compression, which can produce some square artifacts known as blocking. The Deblocking Filter smoothens these edges when decoding the video. The filter is applied on both vertical and horizontal edges. It is a highly adaptive filter, that has 5 strengths. The strength is determined dynamically, depending on the current quantizer, the coding of neighboring blocks and the gradient of the image samples across the boundary.

Arnaldo Azevedo made an SPU implementation of this filter (based on the FFMPEG H.264 decoder) [25] which we use as benchmark. The filter is applied to a full video frame of 320x240 pixels and runs on one SPU. The program reads the input frame from a textfile, filling the different data arrays. This is a very time consuming process in the simulation that takes more than a hour in CellSim, because the data needs some processing on the PPU. To speedup this process, we wrote the data arrays in binary format to a file and used these files to load the different arrays. Now it takes only a few seconds to load the arrays.

6.7.1 Performance Validation

To validate the results on CellSim, the Deblocking Filter kernel was also profiled with SystemSim. Using the CellSim configuration parameters shown in Table 6.20 give the results in Table 6.21.

6.8 Summary

This chapter described the 9 benchmarks we will use to test the branch proposed predictors. For some benchmarks we encountered problems while porting the code to CellSim. We described how we solved those problems and the effect on the results. For each benchmark, the CellSim results were validated with SystemSim. We adjusted the CellSim configuration in order to get results close to the SystemSim results.

Table 6.20: CellSim Configuration for Deblocking Filter.

Parameter	**Value**
BRANCH_MISS_PENALTY	11
HINT_TRIGGER_DELAY	8
IBUFF_CAP	78
FETCH_WIDTH	32
SPU Issue Width	1.5
LDSTQ_SIZE	1
MIN_BRANCH_DIST	8

Table 6.21: SPU Statistics for Deblocking Filter at IBM SystemSim compared to CellSim. The percentages are the number of stall cycles as percentage of performance cycles. The error is relative to SystemSim.

Counter	**SystemSim**		**CellSim**		**Error**	
	Cycles	%	Cycles	%	Cycles	%
performance_cycles	2372687	100.0%	2363310	100.0%	6517	0.3%
fetch_stall_cycles	0	0.0%	7882	0.3%	0	0.0%
dependency_stall_cycles	445359	18.8%	427052	18.1%	-18879	4.2%
channel_blocked_cycles	7020	0.3%	3032	0.1%	-3984	-56.8%
branch_miss_stall_cycles	336835	14.2%	316688	13.4%	3659	1.1%
	Instr.		Instr.		Instr.	
performance_instructions	1923937		1923936		-1	0.0%
branch_instructions	53568		53568		0	0.0%
hint_instructions	10928		10928		0	0.0%
hint_hits	12680		12680		0	0.0%

7 Results and Evaluation

This chapter describes the evaluation of the proposed branch predictors using the benchmarks described in the previous chapter. In the Section 7.1 we describe how the predictors performs with respect to the original SPU without predictor. In Section 7.2 we discuss the energy efficiency.

7.1 Performance

7.1.1 Simple Bimodal Predictor

Figure 7.1 shows the speedup obtained with the Simple Bimodal Branch Predictor (SBP) using the different benchmarks. All speedups are relative to the original SPU architecture. *No hints* are the results when the hint instructions are ignored and no branch predictor is used. *Perfect BP* is when the predictor predicts all branches perfectly. This is simulated by setting BRANCH_MISS_PENALTY=0. It gives an idea what is possible using this predictor, so further results can be compared to this perfect situation. The numbers *64* to *512* are the BHT sizes used.

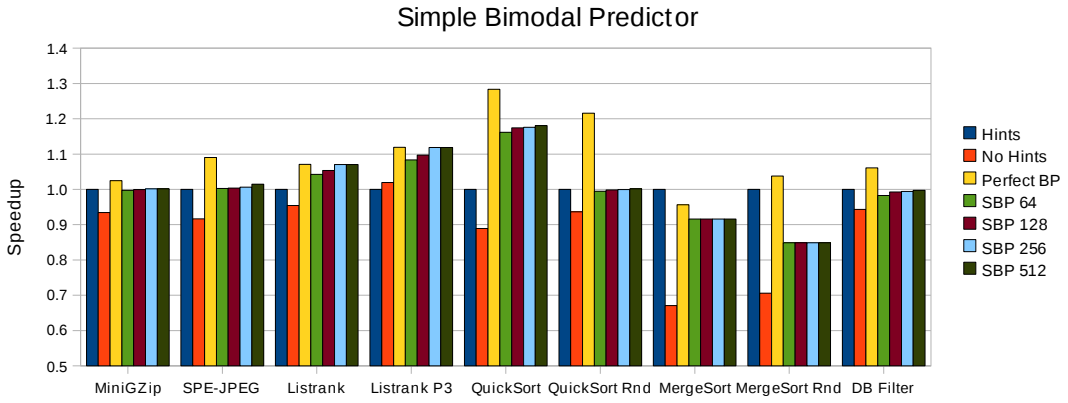

Figure 7.1: Speedup of different benchmarks for the Simple Bimodal Predictor with size 64 to 512.

Ignoring the hint instructions (no hints) gives a significant slowdown for most of the benchmarks. The MergeSort benchmarks have the largest slowdown. This means the hint instructions are very accurate in predicting the branches. ListrankMS-P3 is the only benchmark that shows a speedup when the hints are not used. This is because more than half of the hints are wrong.

Perfect BP shows that speedups from 2.5% for MiniGZip up to 28.3% for QuickSort

are possible. However, for MergeSort a slowdown of 4.4% is the best possible result. The program has 7195 hints that produce 20525 hits, which means one or more hints are reused several times. Using a hint multiple times is very efficient. When a hint is used for the second time, the SPU can continue execution after a taken branch without a branch miss penalty. For the Simple Bimodal Predictor, it still takes seven cycles before the target of a correctly predicted taken branch is fetched from the local store. This is where the hints take advantage.

For MiniGZip, using a Simple Bimodal Branch Predictor instead of branch hints does not make a significant difference in performance.

SPE-JPEG also shows little speedup when the SBP is used. A 512 entry BHT gives a speedup of 1.4%. The maximum speedup that can be achieved is 9.0% which indicates that the bimodal predictor cannot predict the branches very accurate.

ListrankMS benefits more from the branch predictor. A 64 entry BHT gives a 4.3% speedup which increases to 7.0% for BHT larger than 256. When we only look at ListrankMS-P3, these figures increase to 8.3% and to 11.9% for the same sizes. For both, these results are very close to the perfect prediction case, which means that the branches have a very predictable behavior.

QuickSorts has the largest speedup of all benchmarks. It is also the only benchmark that has some performance gain when the BHT size is increased from 256 to 512. For the sequential input, the speedup then becomes 18.1%. However, when the random input is used, there is no significant speedup anymore. For smaller BHT sizes there is even a little slowdown. This shows that the branching behavior of QuickSort is depending on the input. When there is some ordering, the branching is also more predictable. This is because of the while statements that partition the array in a lower en higher part. When a contiguous sequence of elements belongs to the same partition, the program stays in the loop. The branch predictor will be correct then. But if the elements alternately belong to the higher or lower part, the predictor cannot predict this correct.

MergeSort has no speedup, but only slowdown. For all BHT sizes this is the same, namely 8.4%. When the random input is used, it becomes even worse. The slowdown now becomes 15.1%. Like with QuickSort, the branching behavior depends on the input.

The Deblocking Filter also has a small slowdown for al BHT sizes. The slowdown is between 1.9% and 0.3% for BHT size 64 and 512 respectively. When SIMDimizing the SPU code, many if statements are replaced by select statements. So a large part of the branches are due to loops and function calls, which can be predicted quite accurate with hint instructions. The SBP is less accurate for those and thus there are more branch misses. For the other branches, the prediction is just not accurate. For example, there is a branch that alternates between taken and not taken. The predictor cannot predict it right. However, with no branch predictor the SPU is right half of the times (in case the branch is not taken).

Only two of the benchmarks have a speedup when the SBP is used, while others have no significant difference or even a slowdown. Thus the original SPU configuration using hints is a better choice in most cases. If the SBP is used, increasing the size of the BHT above 256 does not significantly improve the performance anymore.

7.1.2 Simple Bimodal Predictor Combined with Hints

The Simple Bimodal Predictor was modified in order to let it take advantage of the hint instructions that are in the code. The implementation makes it possible to combine the information of the hints and the predictor in different ways. The speedup with respect to using only hints for the combinations of hints and a Bimodal Predictor with BHT size 256 described in Table 4.1 are showed in Figure 7.2.

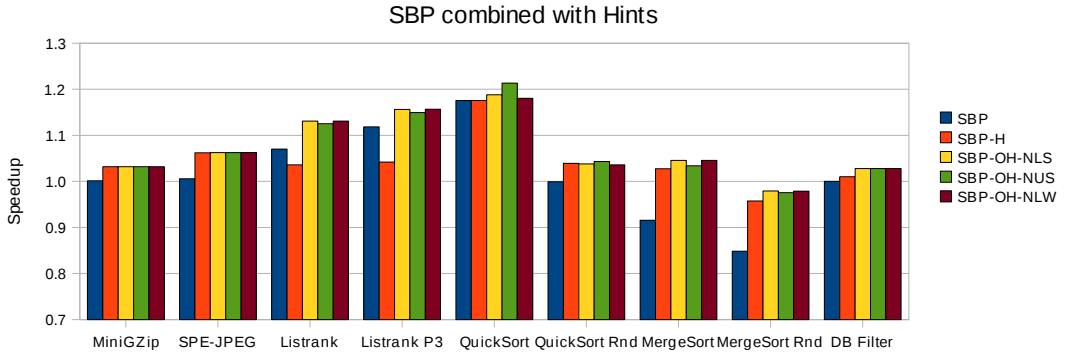

Figure 7.2: Speedup of different benchmarks for the Simple Bimodal Predictor combined with Hints in different ways as described in Table 4.1.

MiniGZip has the same speedup for all four hint policies, namely 3.2%. This is better than the 0.1% speedup of the Simple Bimodal Predictor. It has the same speedup for all the different hint policies, because only a few hints are overruled, namely 143 for the OH-NLS policy. Thus, the speedup is due to correctly predicting non-hinted branches.

SPE-JPEG also has the same speedup for all 4 hint policies, namely 6.2%. This is significantly better than the speedup for the Simple Bimodal Predictor without using hints.

ListRankMS is the only benchmark that has a lower speedup when the hints are used in combination with the SBP than for the SBP alone. This is because most of the wrongly hinted branches are correctly predicted 'not taken' by the predictor. When the predictor prevents hint execution in case it predicts a branch 'strongly not taken', the speedup increases to 13.1% (15.6% for ListrankMS-P3), which is much better than hints only. Also preventing execution of 'weakly not taken' predicted branches does not make a difference. But loading the hint target and not using it, is a little slower.

The performance of MergeSort increases a lot when hints are also used. This results in a speedup for the sequential input of 2.7%. However, the random input still has a slowdown of 4.8%. Overruling hints gives some more speedup. As with ListRank, loading but not using the hint target is the hint policy with the lowest speedup.

Combining hints with the SBP turns the performance slowdown into a speedup of 3.9% for QuickSort with random input. Not loading the hint target is a little slower, especially when 'weakly not taken' predicted branches are also not loaded. But loading the target and not using it gives a higher speedup of 4.3%. This indicates that the hints that are overruled are used several times by the same branch before they are replaced.

With the sequential input, using the hints has no performance improvement. When hints are overruled, the effect is the same for the different hint policies as with the random input, only with larger differences. The highest speedup is 21.3% for the SBP-OH-NUS predictor.

The Deblocking Filter has a speedup for of 2.8% for all combinations where hints are overruled. When the hints are not overruled, there is a the speedup is 1.0%, but that is still faster than without using hints.

In general using hints gives a better speedup than ignoring them, provided that they are overruled by a branch predictor when it predicts not taken. The SBP-OH-NLS predictor that is not loading the hint targets when the prediction is strongly not taken gives the highest speedup in most cases.

Figure 7.3 shows the results of this combination for different BHT sizes. Compared to the results of the Simple Bimodal predictor in Figure 7.1 the speedups are higher, but difference between different BHT sizes is quite similar. Only for the Listrank benchmarks, the speedup difference between BHT sizes is larger, while for QuickSort it is smaller.

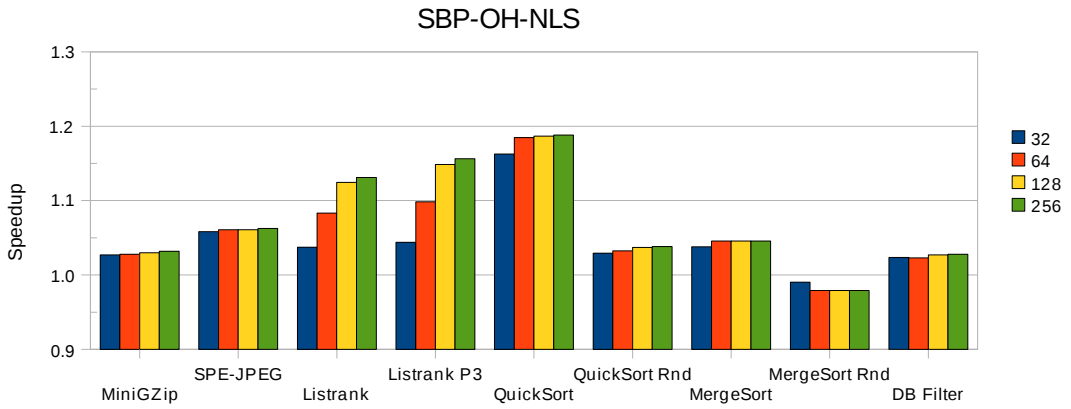

Figure 7.3: Speedup of the SBP-OH-NLS predictor with BTH size 32 to 256 for the different benchmarks.

7.1.3 Aggressive Bimodal Predictor

The Aggressive Bimodal Predictor (ABP) differs from the SBP in the fact that a correctly predicted taken branch instruction could have no branch miss penalty, where the SBP has at least 7 cycles penalty. This should give a higher speedup. However, the hint instructions are ignored so it can not take any advantage from them. Nevertheless, Figure 7.4 that compares the proposed predictors shows that the ABP has the highest speedups for all benchmarks. The speedups for the ABP are depicted is Figure 7.5 for different BHT sizes.

MiniGZip has a small improvement in speedup over both the SBP and the SBP-OH-NLS. For a 256 entry BHT, the speedup becomes 6.1%. This shows that the branching

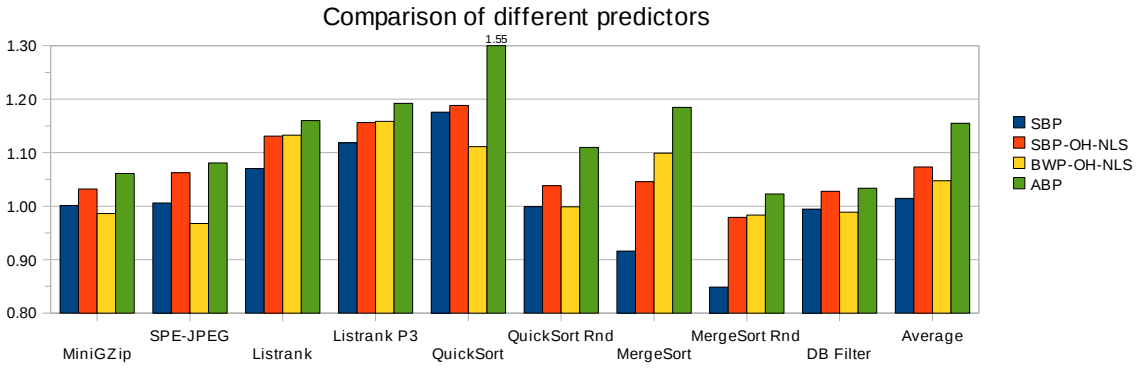

Figure 7.4: Comparison of the proposed branch predictors with 256 entry BHT.

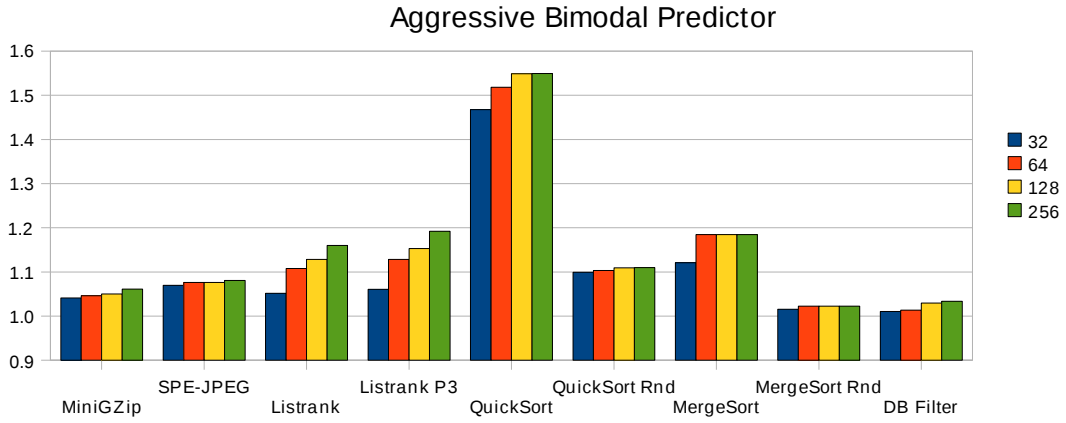

Figure 7.5: Speedup of different benchmarks for the Aggressive Bimodal Predictor with BTH size 32 to 256.

behavior is very variable and thus hard to predict. Decreasing the size of the BHT also decreases the speedup.

SPE-JPEG is also faster, the speedup for a 256 entry BHT is 8.1%. Increasing the size of the BHT above 64 entries gives only a very small performance improvement.

ListrankMS has a larger speedup improvement, especially when a 256 entry BHT is used. The speedup is 16.0% and 19.2% for ListrankMS-P3. As with the other predictors, using a smaller BHT decreases the speedup.

The speedup for Quicksort is about 55%, thanks to the very predictable branching behavior. This speedup is much higher than for the other benchmarks. This is possible because the number of branch miss stall cycles on the original SPU configuration is relatively high compared to the other benchmarks, namely 45%. For the random input, it just 11.0%. However, this is still twice as much as for SBP-OH-NLS.

MergeSort has the same speedup for all BHT sizes, except 32 which is lower. All

are higher then both SBP and SBP-OH-NLS. MergeSort Random has a speedup of the ABP, while it has a slowdown for all other predictors. A 32 entry BHT performs worse than the larger ones.

The Deblocking Filter has a speedup of 3.4% for the 256 entry BHT, which is only slightly better than the SBP-OH-NLS. Decreasing the number of BHT entries to 32 makes the speedup decreases to 1.0%.

For all benchmarks, the Aggressive Bimodal Predictor has the highest speedup. Thus, predicting a branch earlier pays off even without using the hint instructions. However, the branch predictor does a prediction for every instruction, which cost more energy. Also, the instruction line buffer has to be extended with eight lines, which also increases energy consumption. Besides that, it also covers more area than the other predictors because of these extra lines.

7.1.4 Branch Warnings

The results for the branch warning predictor are generated using two configurations: the SBP-H predictor that always uses hint instructions, and the BWP-OH-NLS that overrules the hints using the OH-NLS hints policy. The latter gave the best results for the SBP predictor in Section 7.1.2. The results for both are depicted in Figures 7.6 and 7.7 respectively. MiniGZip, SPE-JPEG, Deblocking Filter and MergeSort show some odd behavior, which is discussed in Sections 7.1.4.1, 7.1.4.2, 7.1.4.3 and 7.1.4.4 respectively.

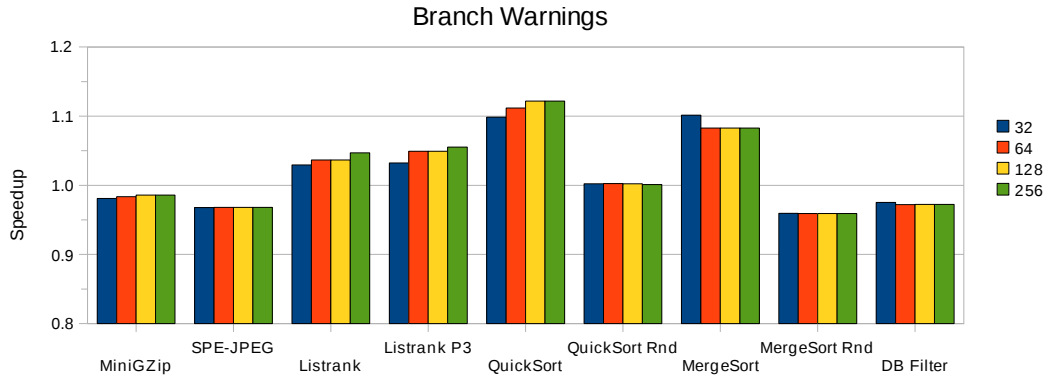

Figure 7.6: Speedup of different benchmarks for the Branch Warning Predictor with BTH size 32 to 256.

ListrankMS performs slightly better using a branch warning predictor with both hint policies, than when the SBP combined with hints is used. The branch prediction using branch warnings takes place earlier than for the SBP, because the branch warnings can be inserted far enough in front of the branch instruction. Thus, there is less penalty for a correctly predicted branch. However, the ABP is still faster.

Both QuickSort benchmarks are slower with the branch warning predictors than for the Simple Bimodal Predictor in combination with hints. Even the Simple Bimodal Predictor is faster when a sequential input is used. Inspecting the programs trace shows

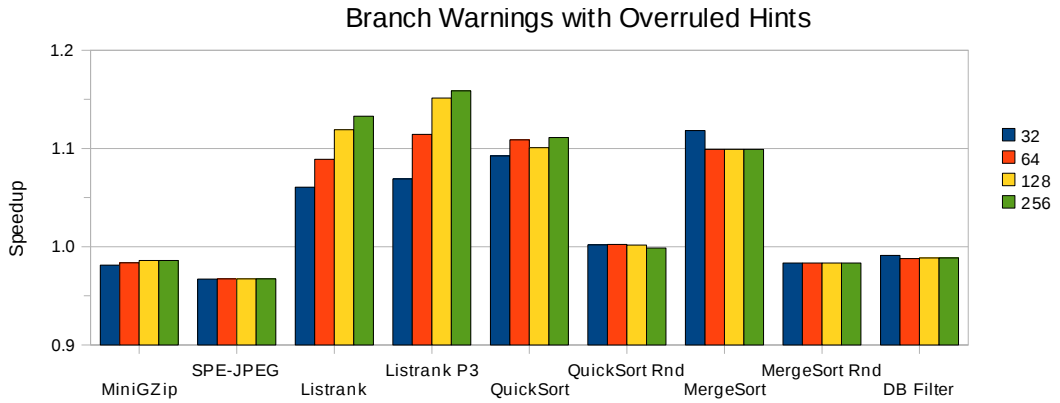

Figure 7.7: Speedup of different benchmarks for the Branch Warning Predictor with OH-NLS hint policy and BTH size 32 to 256.

that a lot of branch misses occur in the second half of the programs execution. There are several pieces of code that have two branches that are separated by only one other instruction. The first branch has a branch warning and when is correctly predicted 'taken', the program continues with no branch miss stall cycles. When the branch is not taken, the second branch instruction is executed. But the second branch has neither a branch warning nor a branch hint. Therefore it is never predicted 'taken'. However, it is taken a lot of times, thus a lot of branch miss stall cycles occur.

7.1.4.1 MiniGZip

MiniGZip has a slowdown of 1.4% for both configurations with a 256 entry BHT, where there was a speedup for all other predictors. When we look at the Branch Statistics in Table 7.1, the number of hint instructions is more than halved with 33K instructions less. Also the number of hint hits is decreased with 20K. This is a indication that the insertion of branch warnings interferes with the insertion of hint instructions.

Comparing the output trace of the MiniGZip program with branch warnings with the original leads to code in Listing 7.1, which is the same piece of code as in Listing 6.2. However, there is a slight difference.

Table 7.1: MiniGZip branch statistics for Branch Warning and Simple Bimodal Predictor both with 256 entry BHT and OH-NLS hints policy.

	BWP-OH-NLS	**SBP-OH-NLS**
Branch Instructions	619437	619438
Hint Instructions	25184	58829
Hint Hits	78406	99490
Branch Warnings	300098	N.A.

Listing 7.1: MiniGZip code with branch warning instead of hint.

```
0x4034:  10000009        hbra    0x24,0
0x4038:  38848285        lqx     $5,$5,$18
0x403c:  4020007f        nop     $127
0x4040:  4020007f        nop     $127
0x4044:  4020007f        nop     $127
0x4048:  4020007f        nop     $127
0x404c:  3b820282        rotqby  $2,$5,$8
0x4050:  1822810c        and     $12,$2,$10
0x4054:  58060603        clgt    $3,$12,$24
0x4058:  20000103        brz     $3,0x8
0x405c:  217ff08f        brnz    $15,0x3ff84
```

Besides the differences in instruction addresses, used registers and the four NOP instructions, the most important difference is the hint instruction. In the original program, this was a hint instruction for the second branch, which is responsible for 19K hint hits. But now it is a branch warning instruction for the first branch. The second branch is now not hinted, neither does it have a branch warning. Thus it is predicted not taken every time which means it is wrong 19K times. If it would be hinted correctly, this could give 342K (19K times 18) less branch miss stall cycles, which would turn the slowdown in a (small) speedup. Probably the same kind of interference applies to the other 3K hint hits that are missing.

As also described in Section 6.5.2.2, part of the code is not compiled with the compiler that inserts branch warnings. Thus there can be branches that are not predicted (which could have been if the modified compiler was used) and thus there can be some extra branch misses with the corresponding branch miss stall penalty.

7.1.4.2 SPE-JPEG

SPE-JPEG also has a slowdown for both hint policies, 3.2% and 3.3% for respectively always using the hints and overruling the hints. Table 7.2 depicts the branch statistics for the BWP-OH-NLS predictor and the SBP-OH-NLS predictor. The first thing noticed is that the number of hints is 36% less for the branch warning predictor. Because of that, the number of hint is 5K or 27% lower. 5K less hint hits means 90K more branch stall cycles, which is about 3% of the total performance cycle count.

Besides that, the number of branch warnings is also quite low. The branch predictor is only used when a branch warning or a hint instruction is executed. Adding those numbers shows that for only 47K branches, which is 49% of the branches, the predictor is used. That explains why this predictor does not have good results.

7.1.4.3 Deblocking Filter

The Deblocking Filter also has a slowdown for both configurations, 2.8% and 2.1% for respectively the BWP-H and the BWP-OH-NLS. Table 7.3 depicts the branch statistics for the Branch Warning Predictor that overrules hints and the SBP-OH-NLS predictor.

Table 7.2: SPE-JPEG branch statistics for Branch Warning and Simple Bimodal Predictor both with 256 entry BHT and OH-NLS hints policy.

	BWP-OH-NLS	**SBP-OH-NLS**
Branch Instructions	95120	95120
Hint Instructions	13758	20961
Hint Hits	13630	18662
Branch Warnings	33280	N.A.

Table 7.3: Deblocking Filter branch statistics for Branch Warning and Simple Bimodal Predictor both with 256 entry BHT and OH-NLS hints policy.

	BWP-OH-NLS	**SBP-OH-NLS**
Branch Instructions	53568	53568
Hint Instructions	8641	10928
Hint Hits	11519	12680
Branch Warnings	18735	N.A.

When branch warnings are used, the number of hint instructions is 30% less. This results in the number of hint being 1K or 10% lower. The extra branch miss stall cycles introduced give a slowdown of 0.7%.

As was the case for SPE-JPEG, the number of branch warnings is also quite low. The branch predictor is only used when a branch warning or a hint instruction is executed. Adding those numbers shows that for only 27K branches or 51% of the branches the predictor is used. That explains why this predictor does not have good results.

7.1.4.4 MergeSort

MergeSort has a larger speedup for the branch warnings predictor than for the SBP in combination with hints, using the same hint policy. The speedup now becomes 9.9% for a 256 entry BHT and overruled hints. MergeSort Random also performs better with branch warnings, although the difference is smaller. But there still is a slowdown of 1.7% in the best case.

The results for MergeSort using branch warnings show some strange behavior: for BHT size 32 the speedup is higher than for BHT size 64 and larger, while the opposite is expected. Further investigation shows that for BHT size 64, 512 branches are wrongly predicted taken. For size 32 there are no wrongly predicted taken branches. The mispredictions are all from one branch instruction: brhnz $2,0x78 at address 0x8FC. Alternately, the branch is taken and not taken. The used bimodal predictor cannot handle that kind of switching efficient and thus the branch is mispredicted every time, which leads to an increase of branch miss stall cycles.

But when a 32 entry BHT is used, the behavior changes. Because there are fewer

Table 7.4: Branch Prediction Statistics for MergeSort with 32 and 64 entry BHT.

	32 entries	**64 entries**
BP Hits	5119	5119
BP Miss	0	512
BHT Entries	23	37
BHT Entries Replaced	4137	32
Hint Hits	20252	20252
Hint Miss	278	278

entries available, more entries are shared between different branch instructions. Now 23 entries are used against 37 for the 64 entry BHT. During the execution of the program, 4137 times an entry replacement occurs (32 for the 256 entry BHT). The branch at 0x8FC is also replaced. The replacement takes place just after the branch was taken to address 0x974. Thus, the next time the branch instruction is executed, no prediction is in the BHT, thus the branch is predicted not taken, which happens to be correct. Therefore there are no branch miss stall cycles. This saves 512*18=9126 cycles, which is about the same as the measured difference.

7.2 Energy Efficiency

IBM has not revealed information about the power dissipation of the SPE. However, a power estimation can be made for a SPE at 3.2GHz made with the 90nm SOI process. Flachs et al [14] made a Voltage/Frequency Schmoo that gives an estimation of the power consumption of the 90nm SPE for different voltage and frequencies. The power is calculated by taking the difference of total power consumption between operating with one and two SPE enabled. Wang [26] tells us that the 65nm Cell processor operates with Vdd = 0.9V. Riley [27] says that the 90nm operates on a 100mV higher power supply setting, thus 1.0V. This Schmoo gives a power of 3W for the SPE at 3.2GHz.

We use CACTI 5.3 rev 174 [28] to create an estimation of the power needed by the BHT. CACTI is a tool for modeling dynamic and leakage power, area and access time of caches and memories. The BHT is quite similar to a direct mapped cache: they are indexed by the address and use tags to identify entries. However, CACTI only supports caches with at least 8 bytes of data per line while our BHT only has 18 bits of data (16-bit BTA and 2-bit prediction). Kahn [13] presents a method to correct for this by scaling the power of the wordline and bitline in the data array with 18/64.

To get a power estimation of a 256 entry BHT, we modeled a cache of size 2048, with 256 lines of 8 bytes. Table 7.5 gives a full overview of the settings used. Table 7.6 shows a selection of the CACTI outputs. The dynamic read power gives the power used when the cache is fully utilized, so there is a read every cycle. The percentages dynamic energy bitlines and wordlines gives the portion of energy that is used by the bitlines and wordlines with relative to the total read energy. The random cycle time is the cycle time

Table 7.5: Used CACTI Settings for 256 8-byte lines cache.

Parameter	Value	Parameter	Value
C	2048	TEMPARATURE	300
B	8	DATA_ARRAY_CELL_DEVICE_TYPE	0
A	1	DATA_ARRAY_PERIPH_DEVICE_TYPE	0
RWP	0	TAG_ARRAY_CELL_DEVICE_TYPE	0
RP	2	TAG_ARRAY_PERIPH_DEVICE_TYPE	0
WP	1	INTERCONNECT_PROJECTION_TYPE	0
NSER	0	WIRE_TYPE_INSIDE_MAT	1
NBANKS	1	WIRE_TYPE_OUTSIDE_MAT	1
TECH	90	REPEATERS_IN_HTREE	1
OUTPUT_WIDTH	18	VERT_HTREE_WIRES_OVER_THE_ARRAY	0
CUSTOM_TAG	1	BROADCAST_ADDR_DATA_OVER_VERT_HTREE	0
TAG_WIDTH	8	MAX_AREA_CONSTRAINT	50
ACCESS_MODE	0	MAX_ACC_TIME_CONSTRAINT	10
PLAIN_RAM	1	MAX_REPEATER_DELAY_CONSTRAINT	10
DRAM	0	PAGE_SIZE	0
OPT_DYN_ENERGY	0	BURST_LENGTH	1
OPT_DYN_POWER	0	INTERNAL_PREFETCH	1
OPT_LEAK_POWER	0	OPT_RAND_CYCLE_TIME	1

Table 7.6: CACTI Results for cache with 256 8-byte lines.

Parameter	Value
Dynamic read power (mW)	26.21400
Standby leakage per bank(mW)	0.49773
Dynamic read energy (nJ)	0.011002
Dynamic write energy (nJ)	0.008260
Perc dyn energy bitlines (%)	14.03230
Perc dyn energy wordlines (%)	5.60833
Random cycle time (ns)	0.41970

used for the modeling.

The results should be corrected for the larger line size. For that, the power used by the bitlines and wordlines is scaled with 18/64. Besides that, the frequency is also incorrect (2.37 GHz instead of 3.2 GHz) is also corrected. Table 7.7 shows that reading a entry from a 256-entry BHT every cycle costs 30.24 mW.

7.2.1 Simple Bimodal Predictor

The power efficiency of the Simple Bimodal Predictor is based on the fact that only a small portion of the executed instructions is a branch instructions. Instead of doing a BHT lookup for every instruction, this is only done for branch instructions. Figure 7.8

Table 7.7: CACTI Results corrected for 256 18-bit entry BHT running at 3.2 GHz.

Parameter	**Value**
Power bitlines + wordlines (mW)	5.15
Line size corrected power bitlines + wordlines (mW)	1.45
Line size corrected dynamic read power (mW)	22.51
Frequency and linesize corrected dynamic read power (mW)	30.24

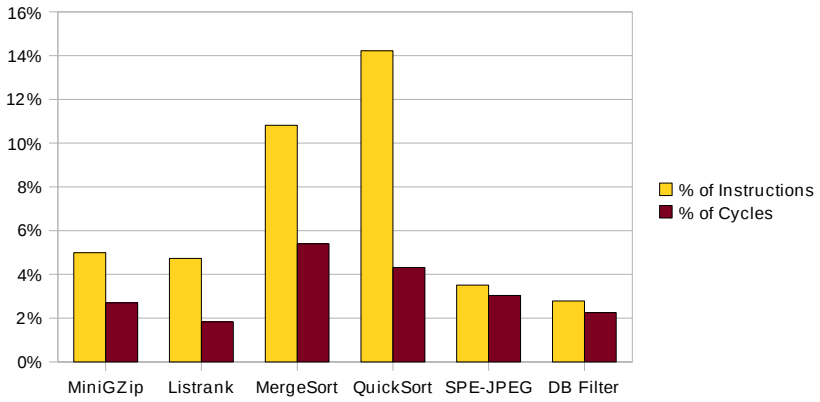

Figure 7.8: Number of branch instructions relative to the performance instruction count and the performance cycle count.

shows that the number of branch instructions relative to the number of performance instructions for the used benchmarks is between 3.5% for SPE-JPEG and 14.2% for QuickSort, or 8.7% on average. This means that the energy needed for determining if a instruction is a branch should be less than 85.8% of the energy to do a BHT lookup in order to be more efficient in worst case. We have no options to determine this, but because the logic for this is quite simple, we assume this is true.

Besides table lookups, the BHT also has to be written/updated when a branch is executed. The number of writes does not change with the proposed predictors. The CACTI results show that a write uses 25% less energy that a read.

The CACTI results are for a BHT that is read every cycle, but this is not true for our implementation. To scale the power result, Figure 7.8 shows the number branch instructions relative to the number of performance cycles. In the worst case (QuickSort), there is a branch instruction in 5.07% of the performance cycles. This means that the power needed for the BHT read and writes is 1.75 * 5.07% * 30.24 = 2.86 mW. Adding the leakage power gives a total of 2.91 mW, which is about 0.1% of the total SPU power.

However we have to add some power for the branch instruction detection. If we assume that detecting a branch instruction costs the same as a BHT lookup, the SPU power increases less then 1% in total.

Also we have to adjust for not using hint instructions. Most of the power is used to

fetch the branch target into the instruction line buffer. However, these instructions also have to be fetched if no hints are used, so not much power is saved. So for simplicity we assume that not using the hints needs no correction in SPU power usage.

The energy used by a program is determined by the product of the power of the SPU and the time it is running. This is called the Energy-Delay product. Less cycles means less energy. With the Simple Bimodal Predictor, the SPU uses 1% more energy. In order to use less energy, the speedup has to be larger than 1%. For the used benchmarks, this is only true for ListrankMS and QuickSort. All others have a lower speedup or even a slowdown. Therefore, overall this is not a efficient solution.

7.2.2 Simple Bimodal Predictor Combined with Hints

For the Simple Bimodal Predictor in combination with hints the power usage is about the same as without hints, because we assume that not using hints would not affect the SPU power.

The energy-delay products are very different story. The SBP-OH-NLS predictor has a speedup larger that 1% for all benchmarks, except MergeSort Random. Thus this combination of the SBP and hint instructions is much more efficient than without hints.

7.2.3 Branch Warnings

The predictor that uses branch warnings only does a BHT lookup when a hint instruction or a branch warning instruction is executed. Because in most cases not all branches have a warning or a hint, and some hints are used multiple times, there are less BHT lookups than branch instructions. In worst case the power is the same as for the SBP, 2.91 mW or 0.1% of the total SPU power.

Executing the branch warnings costs extra energy, but because it also costs extra cycles, we do not have to take that into account. The branch targets that are prefetched are stored in a extra line in the ILB, that will also use some power. However, because it is used even less then the BHT (only for predicted taken branches), we can assume that the extra power is not more than that of the BHT. Thus in total the SPU only needs 0.2% extra power, which is the lowest value of all proposed predictors.

When we look at the energy-delay product, this predictor has the best results for ListrankMS, MergeSort and MergeSort Random. For these benchmarks the speedup is highest of all predictors, however MergeSort Random has still a slowdown which makes the energy-delay product increase. For MiniGZIP and SPE-JPEG the product is increased too.

7.3 Conclusions

In the first part of this chapter we discussed the performance results of the proposed branch predictors using different benchmarks. The results of the Simple Bimodal Predictor are very different for all benchmarks. For two is a speedup, and for three a slowdown. For the other there is little difference. The average speedup is 1.4%. For most benchmarks, having more than 256 entries in the BHT does not give a significant

improvement of the performance. When the hint instructions are also used, the results are much better. If the predictor overrules hints that are predicted 'strongly not taken' the speedup is second best for most benchmarks, up to 19.2% for ListrankMS-P3 and 7.3% on average. As expected the Aggressive Bimodal Predictor performs even better. It has the highest speedup of all predictors, on average 15.5%. The Branch Warning Predictor shows some mixed results. For three benchmarks it performs very good, but for three it is the worst. Nevertheless, with an average speedup of 4.7% it is faster that the SBP. However, these bad results are mainly due to the branch warning placement strategy of the compiler, which can be improved. If that is fixed, this can be the best choice, but for now the Simple Bimodal Predictor that overrules hints using the OH-NLS policy is the best in general. The Aggressive Branch Predictor is faster, but is not a serious candidate because of its complexity, the higher energy consumption, and area usage. This conflicts with the SPU's design characteristics, which are simplicity, low power and area usage.

In the second part of this chapter the energy efficiency was discussed. The additional power needed by the SBP and the SBP-OH-NLS predictor is estimated to be 1% of the 3W total power dissipation of the SPU. The branch warning predictor uses less power, because it does not have to detect branch instructions. When we look at the total amount of energy used for executing the benchmarks using the energy-delay product, the SBP-OH-NLS predictor gives the best results with a reduction 6.6%, mainly because of its performance is highest.

8 Conclusions and Future Work

In this chapter we summarize the work done and draw conclusions. We also do some recommendations for future work in Section 8.2.

8.1 Conclusions

The Cell SPU does not have a dynamic branch predictor, but relies on hint instruction inserted by the compiler to predict branches taken. Because there is a large penalty of 18 cycles for a branch miss, branching can have a large influence on performance.

We proposed four branch predictors to investigate if the branching performance of the SPU can be improved. All are based on a Branch History Table (BHT) that stores the branch target address and a prediction which is made using a bimodal counter.

The first proposed predictor is the Simple Bimodal Predictor(SBP) that does a BHT lookup when an instruction is fetched from the Instruction Line Buffer (ILB). To save energy, this is only done for branch instructions. Detection of a branch instruction is done by pre-decoding the instruction in the ILB. Because of that, a correctly predicted taken branch still has a penalty of 7 cycles.

The SBP ignores hint instructions, but they contain valuable information. Therefore we also made an implementation that uses both hints and a predictor. We combined them using four different hint policies, ranging from always use the hint to ignore the hint when the prediction is not taken.

For the third predictor we introduced branch warnings, that are hint instructions for branches of which the compiler cannot determine the target. The predictor is only accessed when a branch warning or a hint instruction is executed, which makes branch detection in the ILB unnecessary. We tested the predictor using different hint policies.

Finally, we proposed an Aggressive Branch Predictor (ABP) to find out how much speedup is obtainable with only a hardware branch predictor. It is a more complex design and is not supposed to be energy efficient. The branch prediction process is started when instructions are fetched from the local store after a flush, two every cycle. When instructions are issued, the prediction is done for each issued instruction.

The branch predictors are implemented in CellSim, a simulator based on UniSim. Before that, CellSim was extended to support hint instructions and two bugs involving DMA list requests were fixed. To test the performance of the branch predictors we used several benchmarks, which we had to port to CellSim. Before testing, we validated the benchmarks performance on CellSim using IBM SystemSim and adjusted the CellSim settings to get comparable results.

As expected, the aggressive branch predictor has the best performance, with an average speedup of 15.5%. However, it is not a good choice for the SPU because of its complexity. The SBP that uses the OH-NLS hints policy improves performance with an average speedup of 7.3%. The extra power needed is estimated at 1%, thus the energy-delay product decreases by 6.6%. Therefore this is a good extension to the current SPU.

The Branch Warning Predictor that overrules hints (BWP-OH-NLS) currently achieves an average speedup of 4.7%. However, we except the performance to be better than the SBP-OH-NLS predictor, when the compiler is further improved regarding the insertion of branch warnings. The estimated extra power is already lower, which makes the energy-delay product even lower. Thus this is probably the best branch predictor to add to the SPU.

8.2 Future Work

Evaluating the results showed that insertion of branch warnings is not optimal. Currently, the branch warnings are handled like normal hint instructions, only with target 0. However, this interferes with the placement of normal hint instructions and for some benchmark programs the branch warnings prevent a hint to be inserted. In hardware the two are handled differently, therefore the compiler should also do that. In that way, more branches that can have either a hint instruction or a branch warning. Finding an optimal algorithm for inserting hints and branch warnings is needed to get the best out of this predictor.

We used a bimodal counter to calculate the branch prediction because of its simplicity. However there are a lot of other ways to calculate the prediction. Using one of those with the proposed branch predictors is also a good idea for further research.

Bibliography

[1] C. Gou, G. Kuzmanov, and G. N. Gaydadjiev, "Matched sams scheme: Supporting multiple stride unaligned vector accesses with multiple memory modules," Tech. Rep., October 2008.

[2] C. Meenderinck and B. Juurlink, "Specialization of the cell spe for media applications," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, July 2009.

[3] R. Giorgi, Z. Popovic, and N. Puzovic, "Introducing hardware tlp support for the cell processor," in *Proceedings of IEEE International Workshop on Multi-Core Computing Systems*, 2009.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis *et al.*, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep., Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[5] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, "Power issues related to branch prediction," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, 2002.

[6] C. Yang and A. Orailoglu, "Power efficient branch prediction through early identification of branch addresses," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES)*, 2006.

[7] A. Baniasadi and A. Moshovos, "Branch predictor prediction: A power-aware branch predictor for high-performance processors," in *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 2002.

[8] ——, "Sepas: a highly accurate energy-efficient branch predictor," in *Proceedings of the 2004 international symposium on Low power electronics and design (ISLPED)*, 2004.

[9] A. Baniasadi, "Power-aware branch predictor update," in *IEE Proceedings Computers and Digital Techniques*, 2005.

[10] D. Chaver, n. Luis Pi M. Prieto, F. Tirado, and M. C. Huang, "Branch prediction on demand: an energy-efficient solution," in *Proceedings of the 2003 international symposium on Low power electronics and design (ISLPED)*, 2003.

[11] 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.

[12] N. Tomás, J. Sahuquillo, S. Petit, and P. López, "Reducing the number of bits in the btb to attack the branch predictor hot-spot," in *Proceedings of the 14th international Euro-Par conference on Parallel Processing (Euro-Par)*, 2008.

[13] 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.

[14] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee *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.

[15] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins *et al.*, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, 2006.

[16] "Cell Broadband Engine - Programming Handbook - Version 1.1," 2006. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326

[17] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: the end of the road for conventional microarchitectures," *SIGARCH Computer Architecture News*, vol. 28, no. 2, 2000.

[18] *Performance Analysis with the IBM Full-System Simulator - Modeling the Performance of the Cell Broadband Engine Processor*, 2007. [Online]. Available: http://www.ibm.com/developerworks/power/cell/documents.html?S_TACT=105AGX16&S_CMP=LP

[19] 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, pp. 181–188.

[20] "UNISIM." [Online]. Available: http://unisim.org/site/

[21] *Synergistic Processor Unit - Instruction Set Architecture - Version 1.2*, 2007. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326

[22] D. A. Bader, V. Agarwal, and K. Madduri, "On the design and analysis of irregular algorithms on the cell processor: A aase study on list ranking," in *IEEE IPDPS*, 2007.

[23] S. Isaza, F. Sánchez, G. Gaydadjiev, A. Ramirez, and M. Valero, "Preliminary analysis of the cell be processor limitations for sequence alignment applications," in *Proceedings of the 8th international workshop on Embedded Computer Systems (SAMOS)*, 2008.

[24] D. A. Bader, V. Agarwal, K. Madduri, and S. Kang, "High performance combinatorial algorithm design on the cell broadband engine processor," *Parallel Computing*, vol. 33, no. 10-11.

[25] A. Azevedo, C. Meenderinck, B. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of video filtering on the cell processor," May 2008.

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

[27] M. Riley, B. Flachs, S. Dhong, G. Gervais, S. Weitzel, M. Wang *et al.*, "Implementation of the 65nm cell broadband engine," in *Custom Integrated Circuits Conference (CICC)*, Sept. 2007.

[28] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," HP Laboratories, Tech. Rep., 2008.

Curriculum Vitae

Martijn Briejer was born in Voorburg, The Netherlands on the 4th of February 1980. From 1992 to 1998 he did his secondary education at Sint Laurens College in Rotterdam. There he studied at the level of VWO and successfully took the exams in the subjects: Dutch, English, Mathematics B, Physics, Chemistry, Biology and Geography.

After the secondary education he became a student at the Technical University Delft. He enrolled in the faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS). There he got his Bachelor Degree in Electrical Engineering in September 2002.

He chose to continue with the Master Computer Engineering at the same faculty. In November of 2003 he started an internship at DDInfo. There he designed a incremental backup algorithm for the BackupAgent and implemented the server software. After a break, he started in May 2008 with his thesis at the Computer Engineering group of the EEMCS faculty the Delft Technical University with Ben Juurlink as his advisor.

Most of his weekend he is active by Seascouts "De Argonauten" in Rotterdam, where he is the Skipper (team leader). Together with the scouts he does a lot of outdoor activities like sailing, hiking, camping, and climbing. In the winter he likes to go skiing in the Alps.