Branch Penalty Reduction on IBM Cell SPUs via Software Branch Hinting

Jing Lu, Yooseong Kim, Aviral Shrivastava, and Chuan Huang
Compiler Microarchitecture Lab
Arizona State University, Tempe, AZ 85281
{Jing_Lu, Yooseong.Kim, Aviral.Shrivastava, Chuan.Huang}@asu.edu

ABSTRACT

As power-efficiency becomes paramount concern in processor design, architectures are coming up that completely do away with hardware branch prediction, and rely solely on software branch hinting. A popular example is the Synergistic Processing Unit (SPU) in the IBM Cell processor. To be able to minimize the branch penalty using branch hint instructions, in addition to estimating the branch probabilities (which has been looked at before [6, 25, 24]), it is important to carefully insert branch hints. Towards this, in this paper, we i) construct a branch penalty model for compiler, ii) formulate the problem of minimizing branch penalty using branch hinting and iii) propose a heuristic to solve this problem. The heuristic is based on three basic techniques that we introduce in this paper: NOP padding, hint pipelining, and nested loop restructuring. Experimental results on several benchmarks show that our solution can reduce the branch penalty as much as 35.4% over the previous approach.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms

Performance

Keywords

Branch hint, Cell processor, Compiler optimization

1. INTRODUCTION

One of the critical limitations of pipelined modern computer architecture is the branch penalty, which grows larger as the pipeline depths increase. To minimize the impact of branch penalties, target of the branch is predicted, and instructions from there are speculatively fetched. This prediction is typically history based, and implemented in hardware. Because performance of a pipelined processor is critically dependent on the accuracy of branch predictions, many processors use large Branch Target Buffers (BTBs) to store the results of previous branches, and use complex, and often proprietary algorithms to predict the branch target [23, 15].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0715-4/11/10 ...$10.00.

Benchmark	Branch penalty
cnt	58.5%
insert_sort	31.4%
janne_complex	62.7%
ns	50.9%
select	36.2%

Table 1: Branch penalty can be crippling in the Cell SPU in absence of any branch hints.

While branch prediction became the de-facto standard in processor architectures, with the turn of the century, multicores and power-efficiency (MIPS/mW) became an increasingly important consideration in processor design. With the total power budget capped, more cores can only be added by reducing the power and the complexity of each core [9, 11]. Consequently, architects started looking at processor components that could be removed to simplify the cores, yet not lose too much on performance [3]. In the power-efficient IBM Cell Synergistic Processing Units (SPUs) [17], jointly developed by Sony, Toshiba, and IBM, architects decided to remove hardware branch predictor and used software branch hinting in the hope to recover lost performance [22]. This is a significant departure from earlier architectures that supported software branch hinting, e.g., the Sun Niagara [20] and Intel Itanium [14], in the sense that the Cell SPUs do not have any hardware branch predictor, and they rely solely on software branch hints. Table 1 shows the branch penalty (percentage of execution time spent in branch penalty) on some of our benchmark applications. It shows that branch penalty in the absence of any branch hints can be very significant, and inserting branch hints to minimize branch penalty is important for the success of such architectures.

In software branch hinted processors, the application may contain branch hint instructions which indicate that the branch instructions at specified PC addresses will jump to specified target addresses. After executing a hint instruction, the hardware will start to speculatively execute target instructions when the specified branch instruction is executed. For software branch hinting to work best, there are two fundamental considerations: first is to estimate the taken probabilities of branches, and the second is to find the locations in the code for branch hint instructions to mini-

mize branch penalty. The first problem is important because branch hint instructions should be inserted for only heavily taken branches. This problem has been extensively studied [6, 25, 24], but, the second problem, to insert branch hint instructions to minimize branch penalty, is rather unexplored.

Even if we know the taken probabilities of all the branches, minimizing branch penalty by means of branch hint instructions is not trivial. The reasons derive from two constraints of the given architecture. Firstly, for a branch hint to be effective, there must be some separation between a branch and its hint. The hint instruction must be executed several instructions earlier than the branch. Secondly, only a limited number (one for the Cell SPU) of branch hints can be active at any given time. For example, if two branches are too closely located in the control flow, the second branch cannot have enough separation. To hint the second branch, its hint needs to be placed above the first branch, and this will overwrite the hint for the first branch. Thus, hints may conflict with each other, and reduce the achievable benefits.

Towards the problem of minimizing branch penalty in processors with software branch hinting, this paper makes several contributions:

- We construct a branch penalty model for the compiler, in which we express branch penalty as a function of number of instructions between hint and branch instruction, branch probability, and the number of times a branch is executed.
- We present three fundamental approaches to hint branch instructions. i) NOP padding scheme finds out the number of NOP instructions needed between a branch and its hint to maximize profit. ii) Our hint pipelining technique enables hinting branches that are very close to each other, and iii) Our nested loop restructuring technique allows us to change the loop structure to increase the effectiveness of branch hinting.
- Finally, we propose a heuristic that applies the above three methods to the code prudently to minimize overall branch penalty.

Our point of comparison is the branch hint insertion procedure in GCC compiler. It was designed and implemented by Sony and IBM, and included in IBM Cell SDK [13, 1]. We compare the execution time of the benchmarks with our solution to the GCC generated binary. We use static analysis [25] to get the branch probability information of a input program. As shown by experimental results on benchmarks from WCET suite [10] and multimedia loops [19], the proposed heuristic can reduce the branch penalty by 35.4% at maximum and 19.2% on average, at code size increase of merely 3.4%, as compared to GCC.

2. RELATED WORK

Although software branch hinting has been present in processors for a long time, it has not been an active area of research. This is because it has always been in addition to the hardware branch prediction, and in this situation, branch hinting can only improve upon the performance of hardware branch prediction, and the scope of improvement was minimal. However, the Cell processor changes all that. It does not have any hardware branch prediction, and relies solely on software branch hinting to avoid branch penalty. Without any branch hints, severe performance degradation is observed. In such architectures, software branch hinting is no longer optional, but has become mandatory!

In processors with only software branch hinting, branch penalty can be reduced by predication [18] (if supported), i.e., executing both possible execution paths. Loop unrolling [12] can also reduce branch penalty by reducing the number of times branches are executed. Our focus is orthogonal; we intend to reduce branch penalty by hinting the likely-taken branches, by prudent placement of branch hints.

Recently, Briejer et al. [7] studied the energy efficient branch prediction on Cell SPUs by modifying hardware. In their work, the performance and power trade-off of different hardware setups is studied where hardware branch predictor is present in conjunction with software branch hinting. Our techniques, on the other hand are completely in software, and do not require any hardware changes.

There are two main problems in branch hinting to minimize branch penalty: First is to accurately estimate the taken probability of branches, and second is to find prudent placement of branch hints to minimize the penalty. Researchers has been done on estimating taken probabilities of branches. A set of program-based heuristics, especially focused on non-loop branches, was proposed in [6]. Another approach [25] estimates not only branch probabilities but also the execution frequencies of blocks and edges, including function calls, in Control Flow Graphs (CFGs). These techniques are already embedded in GCC compiler. The focus of this paper is the second problem.

GCC compiler, included in IBM Cell BE SDK [13, 1], has a heuristic to insert branch hint instructions to the code. We consider this as the closest related work. It works with a set of principles such as moving hint instructions outside the loops to reduce the overhead of executing hints repeatedly [8], and giving priority to hinting innermost loop branches. GCC suffers from several problems in effectively hinting branches, e.g., if two branches are close to each other, then only one of them is hinted, and in nested loops, typically only the innermost loop branch is hinted.

Our proposed technique alleviates some of the problems of GCC. It carefully analyzes conflicting branches and is able to hint them better through accurate cost functions, and is able to increase the opportunity of hinting low priority branches while keeping all the high priority branches hinted.

3. BRANCH PENALTY MODEL

To implement our technique in a compiler, we need to model the penalty of a branch as a function of i) separation in terms of the number of instructions, ii) the branch probability, and iii) the number of times the branch is executed, which are all the information a compiler can have.

To do this, we conduct several experiments in which we run a synthetic benchmark composed of a branch, and branch hint, separated by a varying number of `lnop` instructions. In each case, we insert some more `lnop` instructions above the hint to keep the total number of `lnop` instructions as 18. We plot the execution time (in cycles) of the benchmark as we change the "separation" between the branch and the hint. The execution time is measured using spu decrementer [12]. Since the granularity of timing measured by spu decrementer is hundreds of cycles, we put the branch and hint in

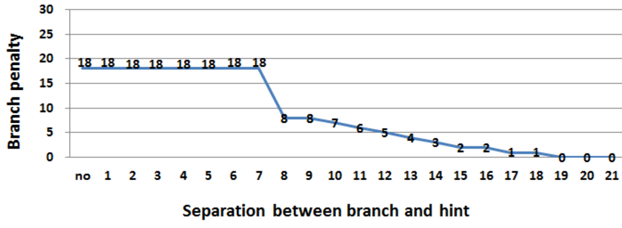

Figure 1: *Branch penalty is plotted as we increase the separation when hint is correct. We need at least 8 instructions for a hint to become effective, and the penalty decreases as separation increases.*

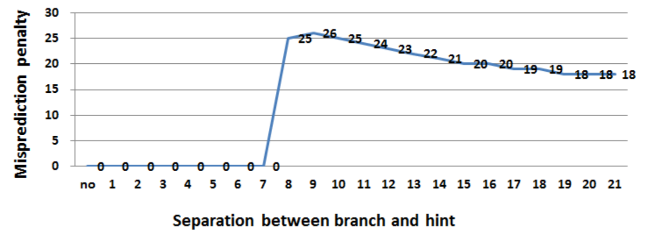

Figure 2: *Misprediction penalty is plotted as we increase the separation. On top of branch penalty, the time to flush pipeline is added resulting larger branch penalty than when hint is correct.*

a loop and execute the loop hundreds times to enlarge the granularity of time measurement.

The reason we insert `lnop` is so that the execution time is not affected by the dual-issue nature of the SPU. SPU is a dual-issue core, and has two unbalanced execution pipelines, named *even* and *odd*, and each of them can execute a disjoint set of instructions. Even pipeline can only execute floating point or fixed point arithmetic operations while odd pipeline can only execute memory, logic, flow-control instructions, including branch and branch hint instruction. Instructions are dual-issued only when i) two instructions are issuable and aligned at an even word address, ii) the first instruction can be executed on even pipeline, and iii) the second instructions can be executed on odd pipeline. There are two NOP instructions, `nop` and `lnop`, in SPUs. `nop` is executed in even pipeline, and `lnop` in odd pipeline. By having only control flow instructions (branch and branch hint) and `lnop`, we effectively make the SPU single-issue.

Figure 1 shows branch penalty plot when the hint is correct (i.e., the branch is taken). When separation is less than eight instructions, the hint is not recognized and we have branch penalty of 18 cycles. After that, the branch waits for the target instructions to be loaded. The penalty decreases with the increase of separation, because executing NOP instructions is now hiding the latency of fetching target instructions. When the separation becomes equal or greater than 19 instructions, the branch penalty can be fully eliminated. The following is our empirical branch penalty model when hint is correct.

$$Penalty_{correct}(l) \approx \begin{cases} 18, & if \;\; l < 8 \\ 18 - l, & if \;\; 8 \le l < 19 \\ 0, & if \;\; l \ge 19 \end{cases} \tag{1}$$

where l denotes the separation in the number of instructions.

Figure 2 shows the same experiment result except that hint was incorrect (i.e., misprediction penalty when the branch is not taken). As expected, when the separation is less than 8, there is no penalty because the architecture assumes branches to be not taken by default. When the separation is greater than 18 instructions, we have 18 cycles of branch penalty due to misprediction. Interestingly, when the separation is between 8 to 18 instructions, the misprediction penalty is greater than 18 cycles and decreases as the separation increases. This means that the branch still waits for target instructions to be fetched, even though the branch is not taken. Thus, branch resolution occurs after target instruction arrives, and this makes incorrect hints more detrimental to performance. Our empirical branch penalty model when hint is incorrect is as follows.

$$Penalty_{incorrect}(l) \approx \begin{cases} 0, & if \;\; l < 8 \\ 36 - l, & if \;\; 8 \le l < 19 \\ 18, & if \;\; l \ge 19 \end{cases} \tag{2}$$

where l denotes the separation in the number of instructions.

Overall, the penalty of a hinted branch is the sum of Equation 1 and 2. Considering branch probability and the number of times the branch is executed, the branch penalty can be calculated as follows.

$$\begin{aligned} Penalty(l, n, p) = & Penalty_{correct}(l) \times np \;\; + \\ & Penalty_{incorrect}(l) \times n(1-p) \end{aligned} \tag{3}$$

where n and p are the number of times the branch is executed, and the branch probability.

4. BRANCH HINTING MECHANISM

The experiments in the previous section give us some intuition about the hardware mechanism of software branch hinting. Figure 3 depicts the mechanism drawn by our inference. The description in this section can perfectly explain the behavior of branch hint instructions.

Just like hardware branch predictors, software branch hinting mechanism also requires a Branch Target Buffer (BTB). When a hint instruction is executed, the BTB entry is first updated, and then target instructions are loaded to the Hint Target Buffer from the specified target address. The hardware usually fetches instructions from Inline Prefetch Buffer which is constantly loaded with the sequential instructions according to PC address. When a branch instruction is fetched, the PC address is compared with the branch address in the BTB entry. If it matches, the instructions are fetched from Hint Target Buffer instead of Inline Prefetch Buffer.

While the actual design may vary, there are three main parameters of any software branch hint implementation.

- ***d*** : the number of pipeline stages where the branch hint is executed and BTB entry is set
- ***f*** : the number of cycles to fetch target instructions
- ***s*** : the size of BTB

The first parameter is d, which is the number of pipeline stages, where the branch hint is executed and BTB entry is set. This implies that, if the separation between the branch hint and branch instruction is less than d cycles, then the

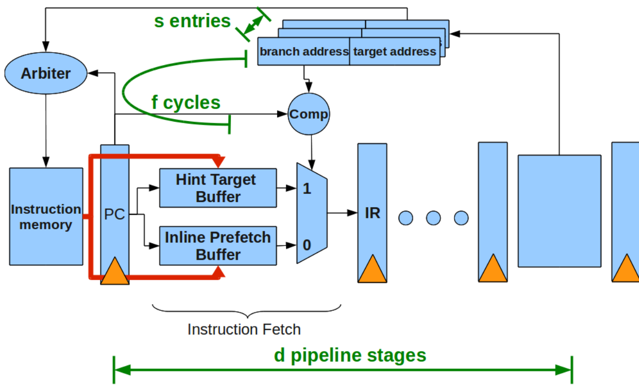

Figure 3: *Software branch hinting is characterized by hint instructions setting the BTB entries. It has 3 key parameters, i) d, the number of pipeline stages where BTB is set, ii) f, the time to fetch target instructions, and iii) s, the number of entries in BTB.*

fetch stage will not even recognize that there is a hint to this branch, and the default prediction (typically, "not taken") will happen. d of SPUs is 8.

The second parameter is f, which is the time (in cycles) to fetch target instructions. After a BTB entry is set, a request is made to the arbiter [12] to fetch the target instructions from memory into the Hint Target Buffer. Note that f may not be statically known, since the delay to get target instructions from the memory depends on the availability of the memory bus. Therefore, in order to completely avoid branch penalty, the separation between a branch hint and branch should be at least $d+f$. This is termed as *separation constraint*, and it is 18 in SPUs. If the branch and branch hint are separated by more than separation constraint, then there is no penalty for a correctly hinted branch. However, if the separation between the branch and branch hint is less than $d+f$, but greater than d, say d', then a correctly hinted branch will incur a branch penalty of $d+f-d'$.

During the hint stall, the branch instruction is stalled before going into execution pipeline. Therefore, even if the hint is incorrect, the comparison between hinted target address and the actually calculated target address, namely branch resolution, can only take place after actually executing the branch instruction. Thus, on top of the branch penalty of 18 cycles, the time to wait for the target instructions to be loaded is added to misprediction penalty. This should be the same as $d+f-d'$ from the above.

The third parameter is s, which is the number of entries in the BTB. A n-entry BTB would imply that n branches can be hinted at the same time. Note that along with the size of BTB, s also impacts the size of Hint Target Buffer, which must be large enough to hold target instructions for all the BTB entries. We expect s to be a small number, to keep the software branch hinting mechanism power-efficient. SPUs have one-entry BTB, making $s = 1$.

5. PROBLEM FORMULATION

The problem of minimizing branch penalty using software branch hinting is to find i) a set of branches to be hinted, and ii) a set of program locations where the hints for those branches should be placed, that will minimize overall branch penalty of the program.

First of all, we need to know branch probabilities and frequencies. This is because, as we discovered in Section 3, when a branch is not taken, hinting the branch will not only increase the instruction count, but also incur a larger branch penalty causing a significant performance degradation. The problem of finding branch probabilities has been well-studied, and improving that state of the art is not the intent of this paper. We use the static estimation technique [6, 25] embedded in GCC compiler, but any branch probability estimation technique can be used.

Even if we know the probabilities, and we have identified branches that benefit by hinting, it is rarely possible to hint all of them. This is primarily because of the separation constraint, which is architecture dependent. In an architecture with s size BTB, only s branch hints can be active at any point of time. In SPUs, only one hint can be effective at any point of execution, which means when two branches are located too close to each other, only one branch can be hinted. To overcome this problem, we will present three methods to enable hinting more branches later in this paper. Since our technique involves restructuring of basic blocks, the control flow of the program may change after applying our technique. However, the program semantic will stay the same. Now, the problem can be formulated as follows.

- **Input:** A program which can be represented in Control Flow Graph, and branch probabilities and frequencies of the branches.
- **Output:** A new program with branch hint instructions. The program may have a different control flow, but the semantic should remain the same.
- **Objective:** Minimize branch penalty, or maximize program performance.
- **Constraint:** For every pair of a hint and branch, separation must be at least d cycles. Also, only s hints can be effective at any point of time, so the lifetime of more than s hints should not overlap. d and s are architecture dependent.

6. OUR APPROACH

In this section, we present three basic techniques which enable us to hint more branches: NOP padding, branch pipelining, and basic block restructuring. We analyze the conditions when the application of each method can be beneficial to performance. Lastly, we will present our heuristic that combines all of them and apply each method prudently.

6.1 NOP Padding

When the enough separation cannot be accommodated, we can insert NOP instructions to artificially increase separation as shown in Figure 4. In the figure, NOP padding increases the separation to eight instructions making the hint to be effective. Let us assume this branch is taken always. Using our branch penalty model, the penalty drops from 18 to 10 cycles. With the help of dual-issue, inserted two `nop`-`lnop` pairs can be executed in two cycles, and thus the total performance improvement is six cycles.

GCC compiler included in IBM Cell SDK also inserts `nop` instructions when a user-specified option is given [1]. We

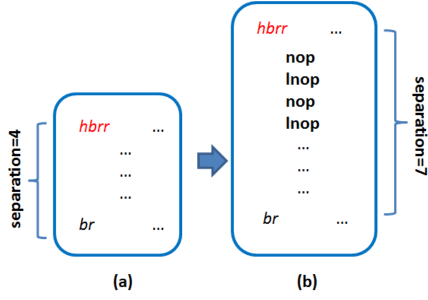

Figure 4: *(a) Before NOP padding, the branch cannot be hinted. (b) NOP padding enables hinting the branch.*

insert both nop and lnop to minimize the overhead of executing additional instructions. When user explicitly specifies the maximum number of nop instructions to be inserted, GCC inserts whenever the branch cannot have enough separation, but in reality NOP padding may not be always profitable. This will be shown in our experimental results.

On the other hand, in our approach, we analyze the performance gain of NOP padding using our branch penalty model. We use NOP padding not only to enable a branch to be hinted but also to increase the performance gain, so called profit, of hinting the branch.

Let l, n, and p respectively denote the original separation before padding, the branch probability and number of times the branch is executed. The branch penalty before applying padding can be calculated as follows.

$$Penalty_{no_pad} = Penalty(l, n, p)$$

Since l is less than the constraint, hint does not work and the penalty is $18np$.

The branch penalty after applying padding is modeled as follows with the separation increased by the number of NOP instructions.

$$Penalty_{pad} = Penalty(l + n_{NOP}, n, p)$$

where n_{NOP} denotes the number of inserted NOP instructions.

Because the branch is taken n times, the hint instruction and the inserted NOP instructions are also taken n times. A pair of nop and lnop instructions can be executed in one cyecle with a help of dual issue. Then, the overhead of NOP padding can be modeled as the following.

$$Overhead_{pad} = n(n_{NOP} + 1)/2$$

Combining all of the above, the performance improvement by NOP padding can be modeled as follows. NOP padding is applied only when it is profitable.

$$Profit_{pad} = Penalty_{no_pad} - Penalty_{pad} - Overhead_{pad} \quad (4)$$

6.2 Hint Pipelining

As shown in Figure 5(a), a compiler may try to hoist the hint for branch b_2 above branch b_1 to increase separation. This will lose any opportunity to hint branch b_2, and this is another common performance limiting factor in GCC. In this case, GCC will simply give up hinting b_1 since b_2 has more priority (Otherwise, GCC would not have tried to hoist the hint in the first place.)

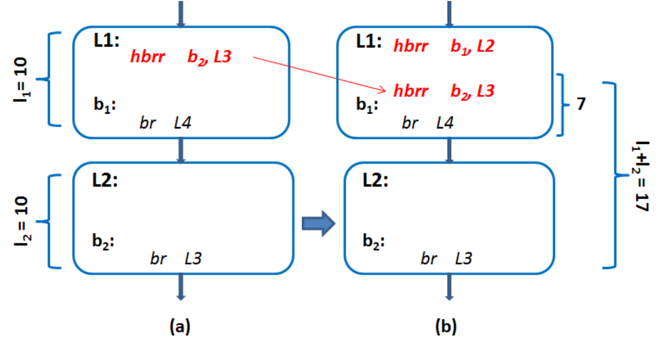

Figure 5: *(a) Before hint pipelining, branch b_1 cannot be hinted due to the hoisted hint for b_2 (b) After hint pipelining, Both b_1 and b_2 are hinted.*

To overcome this problem, let us consider the fact that a hint instruction is not recognized when the separation is less than eight instructions. This gives us an intuition that we can insert multiple hints for multiple branches in a pipelined fashion. Figure 5(b) shows how hint pipelining works. If we place the hint for branch b_2 less than eight instructions ahead of branch b_1, the hint will have not yet recognized when branch b_1 is executed. As a result, we can hint both b_1 and b_2 since the later execution of the hint for b_2 will not affect the previous executed hint for b_1.

We will use the above example to show how to analyze the profit of hint pipelining. In this case, the profit can be modeled as the decrease of branch penalty of newly hinted branch b_1 minus the possible increase of branch penalty of b_2. Let l_x denote the number of instructions in basic block Lx. The path from $L1$ to $L2$ is taken only when the branch b_1 is not taken. Thus, when the branch b_1 is taken, the branch b_2 is not hinted. The branch penalty before applying hint pipelining can be modeled as sum of the penalty of two branches as follows, and the penalty of not hinted branch is modeled as the case when separation is zero.

$$\begin{aligned} Penalty_{no_pipeline} = & Penalty(0, n_1, p_1) \ + \\ & (1 - p_1) \cdot Penalty(l_1 + l_2, n_2, p_2) \ + \\ & p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

where p_x and n_x denote the branch probability of branch b_x and the number of times b_x is executed.

After hint pipelining, both b_1 and b_2 will be hinted. The maximum possible separation for the hint for b_2 is decreased from $l_1 + l_2$ to l_2, which possibly increases branch penalty of b_2, but we can hint another branch b_1 instead. Since our heuristic starts inserting hint instructions from bottom basic blocks, when this analysis is being done, the hint for branch b_1 is not yet inserted. We always assume that b_1 will be hinted at the top of $L1$, even though it can be hinted farther above, possibly reducing more branch penalty. The penalty after applying hint pipelining is modeled as follows.

$$\begin{aligned} Penalty_{pipeline} = & Penalty(l_1, n_1, p_1) \ + \\ & (1 - p_1) \cdot Penalty(7 + l_2, n_2, p_2) \ + \\ & p_1 \cdot Penalty(0, n_2, p_2) \end{aligned}$$

Note that is only applied when $l_1 \geq 8$. The above calculation is an example when a hint is hoisted to the immediate

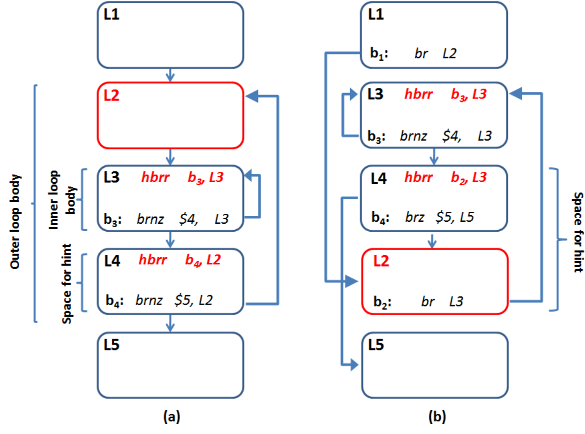

Figure 6: *(a) Before nested loop restructuring, the separation for b_4 is limited to l_4. (b) After nested loop restructuring, the outer loop branch is changed to unconditional branch b_2, and the separation is increased to $l_2 + l_4$.*

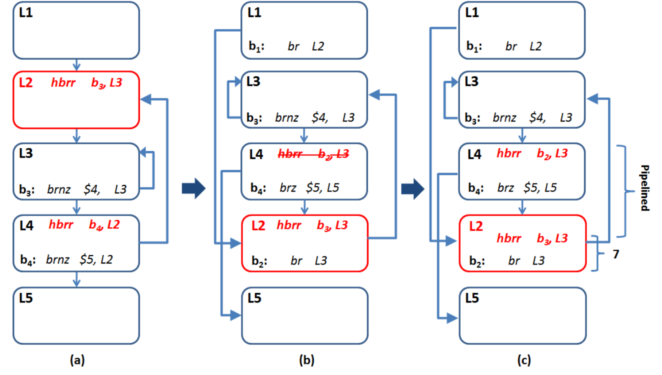

Figure 7: *(a) Hint for b_3 can be hoisted into L2. Branch b_4 can be hinted if l_4 is at least eight instructions. (b) After restructuring, the hint for b_3 cancels the hint for b_2. (c) Pipelining is applied and both branches can be hinted.*

predecessor. A similar analysis can be done to any other cases.

The overhead of hint pipelining is the number of times the hint instruction for branch b_1 is executed. When the hint is in basic block Lx, it is executed n_x times. Then, the overall overhead is the difference of execution counts as shown below.

$$Overhead_{pipeline} = n_1$$

Hint pipelining is applied only when the overall profit of it is greater than zero, which can be modeled as the following.

$$Profit_{pipeline} = Penalty_{no_pipeline} - Penalty_{pipeline} - Overhead_{pipeline} \quad (5)$$

6.3 Nested Loop Restructuring

The branch penalty from loops is important, because even a small penalty can be accumulated for the whole iteration and affect performance significantly. In this section, we present a method specially developed for nested loops. This technique is motivated by our observation that usually in nested loops, only innermost loop branch can be hinted, and the outer loop branch cannot be hinted due to separation constraint.

As summarized in Figure 6, we can change the structure of nested loop so that the space to insert a hint for the outer loop branch[1] is enlarged. In Figure 6(a), let us suppose the size of basic block $L4$ is too small to hint the branch b_4. Figure 6(b) presents our solution in which basic block $L2$ is moved after $L4$, and two unconditional branch b_1 and b_2 are introduced. Also, the target address of branch b_4 is changed to $L5$, and the branch condition is flipped. This technique is applied before any hints are inserted into the code, and here the hint for b_3 is assumed to be placed in $L3$.

Let us consider the same example to analyze the profit of nested loop restructuring. Before applying restructuring, the overall branch penalty is the sum of branch penalties of b_3 and b_4. In this example, l_4 is smaller than eight instructions, so the branch b_4 will not be hinted.

$$Penalty_{no_restructure} = Penalty(l_3, n_3, p_3) + Penalty(l_4, n_4, p_4)$$

After applying restructuring, the outer loop branch is changed to unconditional branch b_2 and it has separation of $l_2 + l_4$. We may get more profit from this, but this introduce branch b_1 which will be taken only once when entering the loop. Also, the branch condition of b_4 is changed, so it is taken only once when exiting the loop. We assume that b_1 and b_4 are not hinted incurring 18 cycles of penalty for each. The penalty becomes the sum of branch penalties of b_1, b_2, b_3, and b_4. Note that the path probability for $L4$ to $L2$ is one since the branch will always fall through except when the loop terminates.

$$Penalty_{restructure} = 18 + Penalty(l_2 + l_4, n_2, p_2) + Penalty(l_3, n_3, p_3) + 18$$

The overhead of this technique is the difference of the numbers of times hint instructions are executed. In this particular example, the hint for b_4 could not be inserted at first due to separation constraint, but now it is inserted into $L4$. However, in general, the nested loop restructuring can be used to improve the profit of b_4 even if l_4 is greater than eight instructions. In this case, the overhead is considered as zero because the hint instructions are not moved to other basic blocks.

$$Overhead_{restructure} = \begin{cases} n_4, & if\ l_4 < 8 \\ 0, & otherwise \end{cases}$$

Nested loop restructuring is only applied when the overall profit of it is greater than zero, which can be modeled as the following.

$$Profit_{restructure} = Penalty_{no_restructure} - Penalty_{restructure} - Overhead_{restructure} \quad (6)$$

Note that in the above example, without loss of generality $L3$ can denote a loop body containing multiple basic blocks. This is because the intention of nested loop restructuring is to give more separation to outer loop branch, and the inner loop is not affected. For the loop body which does

[1] Throughout the paper, we assume that loop branches are always at the bottom of the loop body.

not have any likely-taken branches (e.g. function call or if-then-else), we hoist the hint for the loop branch to the loop-initialization block, which is executed only once. This is to reduce the overhead of repetitive execution of the hint instruction. Figure 7(a) shows an example where the hint for inner loop can be hoisted to outer loop body. After applying restructuring, the hint for b_3 is hoisted to $L2$ and overwrites previously inserted hint for b_2, as shown in Figure 7(b). Instead of canceling the hint for b_2, we can apply pipelining to hint both branches. Figure 7(c) shows the final solution. We check the structure of the inner loop body, and if the hint can be hoisted, we assume that pipelining will be applied later. To determine its profitability, a similar analysis to the above can be done assuming the hint for b_3 will be placed seven instructions above b_2.

This abstraction of loop body enables us to apply this technique to all kinds of loop nests. For loop nests whose depth is more than two, this technique is recursively applied from the innermost loop to the outermost loop. For example, let us suppose we have three loops $L1$, $L2$, and $L3$. $L1$ is the innermost loop, and $L3$ is the outermost loop. $L1$ and $L2$ are first considered as restructuring candidates, and we check the profit of restructuring two loops. If those two loops are reordered, they can be considered as one loop body. Then, either the reordered loop body or $L2$ can be considered as restructuring candidate with the next outer loop $L3$. Also, if there is more than one inner loops, all of the inner loops can be considered as one loop body. This restructuring may result in severe instruction cache misses in conventional machines, but it is not the case in software branch hinting because instructions are explicitly prefetched by branch hint instructions.

6.4 Our Branch Penalty Reduction Heuristic

Given all methods and their applicable conditions we discussed above, now we present a heuristic that combines all of them. It requires the information of all nested loops, the branch probabilities, and the number of times each branch is taken as input.

Algorithm 1 shows the pseudocode of our heuristic. The procedure starts with applying nested loop restructuring to all loop nests to increase the possible separation for loop branches. Then, it starts inserting hint instructions from the bottom basic block. For a branch, the procedure tries to hoist its hint to the predecessor basic blocks, scanning predecessor basic blocks recursively. If there is a branch in the predecessor and it is likely-taken, it stops going into predecessors and returns the current basic block. Then, the procedure insertHint() inserts a hint instruction in the current basic block. It checks the separation and applies NOP padding and hint pipelining when applicable.

Algorithm 1 *Our branch penalty reduction heuristic.*

```
Apply nested loop restructuring for all loop nests;
b = last basic block in the program;
while basic block b is not the first basic block do
  h = b;
  while Basic block h is not the first basic block do
    if Branch in h->predecessor is likey-taken then
      break;
    end if
    h = h->predecessor;
  end while
  insertHint(b, h);
  if b != h then
    b = h;
  else
    b = b->predecessor;
  end if
end while

insertHint(b, h): Insert a hint instruction for the branch
in basic block b into basic block h.
{
if h contains a branch then
  if Pipelining is profitable then
    Insert a hint instruction for the branch in b in a
    pipelined mode;
  else
    if NOP padding is profitable then
      Insert as many NOP instructions as it is profitable;
    end if
    Insert a hint instruction for the branch in b;
  end if
else
  if NOP padding is profitable then
    Insert as many NOP instructions as it is profitable;
  end if
  Insert a hint instruction for the branch in b;
end if
}
```

7. EXPERIMENTAL RESULTS

The effectiveness of the proposed heuristic is validated using various benchmarks from Multimedia loops [19] and WCET benchmarks [10]. Our baseline is GCC compiler [1], which is included in IBM Cell SDK [13]. It has a heuristic that inserts branch hint instructions to the code, which is designed and implemented by the manufacturer. All benchmarks are compiled with O3 optimization level. To measure the performance and the branch penalty of the program, the cycle accurate IBM SystemSim Simulator for Cell BE [2] is used. As library functions (e.g. printf()) are not changed, all the measurements are done only on user codes. Branch probabilities and the cyclic frequencies of branches are obtained by a static analysis [6, 25], which is also implemented in GCC.

Figure 8 shows the percentage of branch penalties in the total program execution cycles after GCC inserts hints. We divide the benchmarks into two groups 'high' and 'low' according to the percentage of branch penalty in the total execution time. The benchmarks which have more than 20% of branch penalty are grouped as 'high', while the others fall under the group 'low'.

7.1 Branch Penalty Reduction

The effectiveness of our heuristic can be shown as the reduction of branch penalty after applying our heuristic. Figure 10 shows the reduction in branch penalty cycles after applying our heuristic, compared to the GCC-inserted hints. Overall, we can reduce average 19.2% of the branch penalty more than GCC. Since we insert NOP instructions through our NOP padding technique, we consider the increased NOP cycles as part of branch penalty. SystemSim simulator can output NOP cycles separately as well as branch penalty cy-

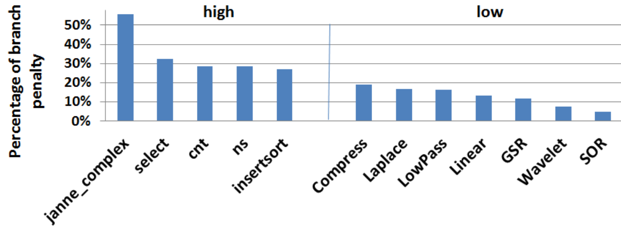

Figure 8: *The percentage of branch penalty in the total execution cycles after GCC inserts hints into the program. Benchmarks are grouped into two groups 'high' and 'low' according to the percentage.*

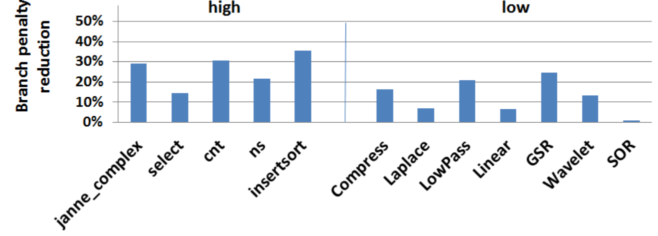

Figure 10: *Reduction of branch penalty is 35.4% at maximum and 19.2% on average.*

cles, and branch penalty in our results is the summation of branch penalty cycles and increased NOP cycles.

The proposed heuristic works more effectively for the benchmarks with deeply nested loops, such as **janne_complex**, **cnt**, **insertsort** and **ns**. As shown in Figure 8, GCC cannot reduce the branch penalty effectively in those benchmarks, and all of them fall under 'low' group. Figure 9 compares the code change in a deeply nested loop in benchmark **ns** after GCC and our heuristic. Loop branches are shown in solid arrows, while others are shown in dotted arrows. GCC can only hint the loop closing branch for the innermost loop, and because of limited basic block sizes, all other loop branches cannot be hinted. Our technique, on the other hand, can hint all of the loop branches.

Even with our technique, the highest reduction of stall due to branch penalty is around 35 percent. There are several reasons why the branch penalty cannot be completely eliminated. Firstly, Not all branches can be hinted because only one hint can be active at a time. When two branches are located too close to each other, only one of them can be hinted. Even though our techniques can enlarge the possible separation to enable more branches to be hinted, they cannot be applied to every case. This is because each technique is applied only when it is profitable. Unless two or more branch hints are allowed to be active at a time, this problem cannot be ultimately solved.

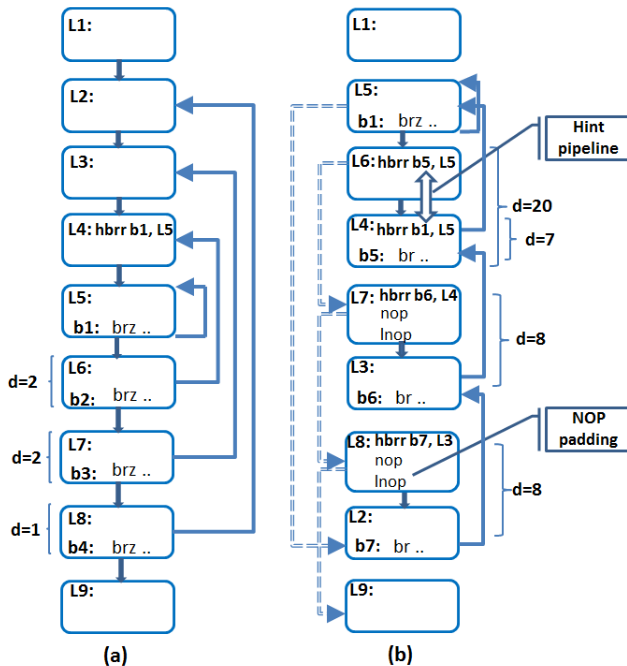

Figure 9: *(a) GCC can only hint the innermost loop branch. (b) The proposed technique can hint all of the four loop branches.*

The other reason is that hint instructions are inserted in compile time and cannot be easily changed in run time. In other words, branch hinting works as a static branch predictor, while most of the branches are dynamically decided to be taken or not. Even though the penalty can be effectively avoided when branch is taken, there is still misprediction penalty when the branch is not taken. Thus, unless a branch is heavily taken, to hint the branch may not be always profitable. A typical example is "if-then-else" branches in a loop. The worst case scenario is when the branch is taken for the half of the time. Penalty always exists whether or not we hint the branch, as long as the hint is static. If the compiler assigns the more-likely-taken execution path as a fall-through path, the penalty of "if-then-else" branch can be effectively avoided [12], but not completely. As it is inside a loop, the penalty gets accumulated and eventually limits the performance. Moreover, the accuracy of branch probability information can be another limiting factor. Branch probabilities affect the decision of which branches should be hinted, and we rely on static analysis to obtain branch probabilities, which may not be very accurate. Use of profile information may be helpful to improve the result. But improving the prediction of branch probability is not within the scope of this paper.

7.2 Effectiveness of Our NOP Padding

GCC also has a mechanism in which `nops` are inserted between a branch and its hint in order to increase the separation and thus the chance of hinting. However, it has no automatic way of determining how many NOPs to insert, and when compiled with "-mhint-max-nop=n", GCC [1] will insert at most n `nops` to ensure the separation is at least eight instructions. In comparison, our scheme automatically finds out the number of NOPs to be inserted, to maximize profit. Figure 11 compares the performance of our NOP Padding approach with that of GCC with $n = 0, 4$, and 8. $n = 0$ means no NOPs will be inserted. Note that among the GCC schemes, sometimes inserting NOPs even decrease the performance. This is because GCC does not have any profitability analysis to find out the number of NOPs to be inserted. Another advantage of our technique is that while GCC only inserts `nops`, while we insert `nop` and `lnop` pairs.

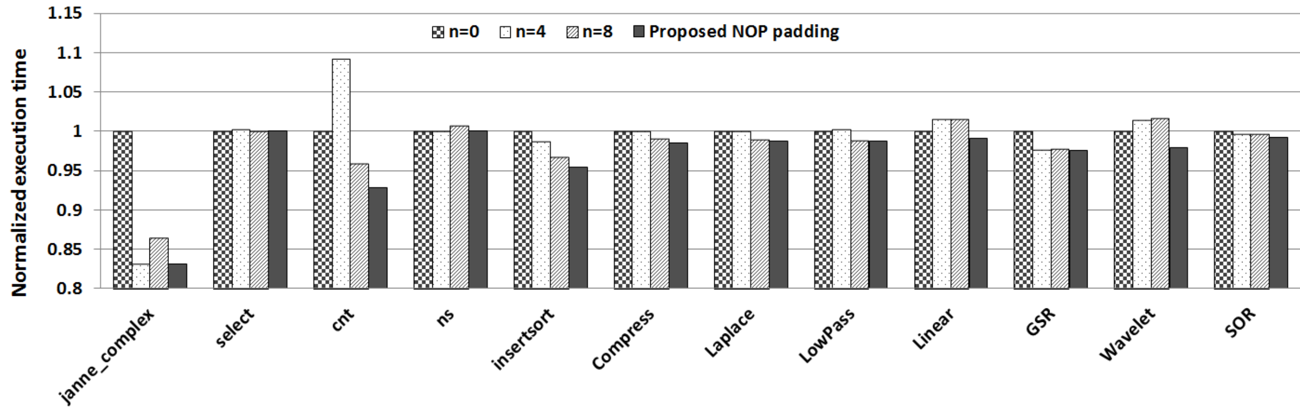

Figure 11: *Execution time comparison between our NOP padding technique and GCC's "-mhint-max-nops" option. In all benchmarks, our technique outperforms GCC. GCC even results in performance degradation for several benchmarks.*

By doing this, we benefit from the dual-issue nature of SPU. Even when two approaches insert the same number of NOP instructions, the performance penalty of our approach is half as that of GCC. Consequently, the performance improvement of our technique always surpasses the one of GCC.

This prudent insertion of NOP instructions is also important in the context of static code size increase. IBM Cell BE is a limited local memory architecture, and each SPU can only access its local store which is of 256 KB. Code, global data, and all dynamic data such as stack and heap data reside in the local store. There have been dynamic management schemes [21, 16, 4, 5] for code and data. However, since all of data and code share the same local store, the increase in code size imposes more burden to those management schemes and thus increases the overhead of them. Therefore, it is important not to increase the code size too much. Note that this is static code size which affects the executable file size, and the dynamic code size increase overhead was already considered and included in the branch penalty reduction results. The average code size increase is merely 3.4%, while GCC incurs 11.7% code size increase with the "-mhint-max-nop=8" option.

7.3 Performance Improvement

The reduction in branch penalty cycles will improve program performance, and the amount of performance improvement depends on the percentage of branch penalty in the total execution time.

Figure 12 shows the performance improvement for each benchmark, and as expected, benchmarks in 'high' group show more performance improvement than those in 'low' group, with the peak speedup of 18%. This is natural in the sense that higher proportion of branch penalty makes them more susceptible to performance improvement via branch penalty reduction. However, benchmark **select** has the second highest branch penalty percentage but shows the lowest speedup in the 'high' group. This is because it has multiple "if-then-else" branches in loops, whose penalty cannot be effectively avoided by software branch hinting as mentioned in the previous section. Though the benchmarks in 'low' group show relatively low speedup, it does not mean that our technique is not effective for those benchmarks. Our technique can reduce over 25% of the branch penalty for the benchmark **GSR**, but it is not fully reflected as reduction in execution time because its percentage of stall due to branch penalty is too low.

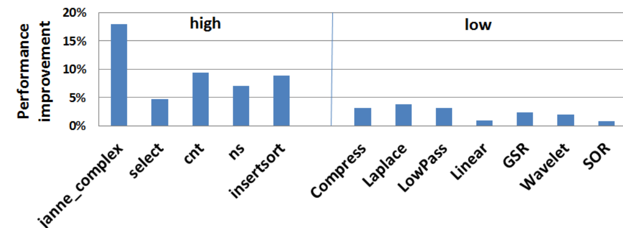

Figure 12: *Performance improvement obtained with the proposed heuristic is as much as 18%.*

An important aspect of our technique is that our heuristic never results in a performance decrease. This is because every step of our technique involves profitability analysis. This guarantee, combined with the fact that the code size increase by our technique is minimal, we argue that it is always beneficial to apply our branch hinting heuristic.

8. CONCLUSION AND FUTURE WORK

Multi-cores and power efficiency have been continuously driving modern processor design. As a result, many complex architectural components are being removed from hardware and required to be implemented in software instead. IBM Cell SPUs removed branch predictor and introduced software branch hinting. Due to a huge branch penalty, branch hint instructions are crucial for performance optimization.

In this paper, we propose a heuristic algorithm to reduce branch penalty using software branch hinting. The algorithm is based on our proposed branch penalty model and three basic techniques: NOP padding, hint pipelining, and nested loop restructuring. The branch penalty model helps us to estimate the branch penalty, and those techniques not only enable more branches to be hinted, but also reduce more branch penalty. Our experimental results for WCET [10] and multimedia benchmarks [19] show that our approach can reduce the branch penalty by 35.4% at maximum and 19.2% on average more than GCC's heuristic which is designed and implemented by the manufacturer.

There are several tasks to be done as future work. As the architectural features such as BTB size are ultimate factors that limit the effectiveness of software branch hinting, the SW/HW codesign approach to find the optimal BTB size is needed to improve the effectiveness and applicability of software branch hinting. A software technique to make a dynamic branch predictor using software branch hinting will be useful if the overhead of dynamic prediction can be kept minimal.

9. ACKNOWLEDGEMENT

This work was partially supported by funding from National Science Foundation grants CCF-0916652, CCF-1055094 (CAREER), Raytheon, NSF I/UCRC for Embedded Systems (IIP-0856090), Intel, Microsoft Research, SFAz and Stardust Foundation.

10. REFERENCES

[1] GNU Toolchain 4.1.1 and GDB for the Cell BE's PPU/SPU. `http://www.bsc.es/plantillaH.php?cat_id=304`.

[2] IBM Full-System Simulator for Cell BE. `http://www.alphaworks.ibm.com/tech/cellsystemsim`.

[3] A. Agarwal and M. Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 750–753, New York, NY, USA, 2007. ACM.

[4] K. Bai and A. Shrivastava. Heap Data Management for Limited Local Memory (LLM) Multi-core Processors. In *CODES+ISSS '10: Proceedings of the 23th international symposium on System Synthesis*, New York, NY, USA, 2010. ACM Press. ISBN.

[5] K. Bai, A. Shrivastava, and S. Kudchadker. Stack Data Management for Limited Local Memory (LLM) Multi-core Processors. In *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors (ASAP)*, 2011.

[6] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of PLDI*, pages 300–313, New York, NY, USA, 1993. ACM.

[7] M. Briejer, C. Meenderinck, and B. Juurlink. Extending the Cell SPE with energy efficient branch prediction. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 304–315, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[9] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[10] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[11] H. Hofstee. Power efficient processor architecture and the Cell processor. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 258–262, 2005.

[12] IBM. Cell Broadband Engine Programming Handbook including PowerXCell 8i. `https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/7A77CCDF14FE70D5852575CA0074E8ED`.

[13] IBM. IBM Cell SDK 3.1. `http://www.ibm.com/developerworks/power/cell`.

[14] Dual-Core Intel Itanium Processor 9000 and 9100 Series. `http://download.intel.com/design/itanium/downloads/314054.pdf`, 2007.

[15] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 197, Washington, DC, USA, 2001. IEEE Computer Society.

[16] S. c. Jung, A. Shrivastava, and K. Bai. Dynamic code mapping for limited local memory systems. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 13 –20, 2010.

[17] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49:589–604, July 2005.

[18] J. Kalamatianos and D. R. Kaeli. Improving the accuracy of indirect branch prediction via branch classification. *SIGARCH Comput. Archit. News*, 27(1):23–26, 1999.

[19] D. Kolson, A. Nicolau, and N. Dutt. Elimination of redundant memory traffic in high-level synthesis. *IEEE Trans. on Comp-aided Design*, 15:1354–1363, 1996.

[20] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[21] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping for Scratchpad Memories. In *HIPC 2008 :International Conference on High Performance Computing*, pages 569–582, 2008.

[22] B. Sinharoy and S. W. White. Use of software hint for branch prediction in the absence of hint bit in the branch instruction. `http://www.freepatentsonline.com/6971000.html`.

[23] A. S. Stephen, S. Felix, V. Krishnan, and Y. Sazeides. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *in 29th Annual International Symposium on Computer Architecture*, pages 295–306, 2002.

[24] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 85–96, New York, NY, USA, 1994. ACM.

[25] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, New York, NY, USA, 1994. ACM.