計算機アーキテクチャ 118－4
(1996. 5. 16)

Speculative Execution by Compiler Supported Branch Prediction Hardware

Tetsuo Hironaka† Ashok Halambi‡ Alex Nicolau‡ Nikil Dutt‡

†: 広島市立大学 情報工学科,
〒731-31 広島市 安佐南区 沼田町 151-5
E-mail: hironaka@ce.hiroshima-cu.ac.jp

‡: Department of Information and Computer Science,
University of California, Irvine CA 92717, USA
E-mail: {ahalambi, nicolau, dutt}@ics.uci.edu

本報告ではコンパイラによって制御されるハードウェア分岐予測機構を用いて効率的な投機的命令実行を実現する手法を示す.

スーパスカラ・プロセッサ, スーパパイプライン・プロセッサなどの命令レベル並列処理を利用するプロセッサにおいて命令多重度を充足するためのオブジェクト・コード最適化が大きな問題である. そして, オブジェクト・コード最適化にもっとも障害となっているのが制御依存による先行制約である. 投機的命令実行は制御依存の影響を軽減するための有効な手段であるが, それは正しい命令を選択し投機的に実行したときのみである.

本報告ではソフトウェアにより制御可能な分岐予測機構をプロセッサ内に導入することで非常に高い確度で有効な投機的命令実行が可能であることを示す.

Speculative Execution by Compiler Supported Branch Prediction Hardware

Tetsuo Hironaka† Ashok Halambi‡ Alex Nicolau‡ Nikil Dutt‡

†: Department of Computer Engineering,
Hiroshima City University, Hiroshima 731-31, JAPAN
E-mail: hironaka@ce.hiroshima-cu.ac.jp

‡: Department of Information and Computer Science,
University of California, Irvine CA 92717, USA
E-mail: { ahalambi, nicolau, dutt }@ics.uci.edu

This paper describes an effecttive method for speculative execution by compiler supported branch prediction hardware. For processors architecture which count on instruction level parallelism, such as superscalar processors, and superpipeline processors, optimizing object code to extract more instruction level parallelism is a big problem. Especially the control dependency inside the object code makes this problem worse. Speculative execution is an effective method to handle the control dependency to achieve more instruction level parallelism. But speculative execution is effective only when the correct operations are selected for speculation.

This paper presents a method to select correct operations to speculate in high possibility, in use of compiler supported branch prediction unit.

Keywords: speculative execution, branch prediction, compiler, superscalar, VLIW

1 Introduction

For processors architecture which its performance depends on the amount of instruction level parallelism, such as superscalar processors, and superpipeline processors, it is important to optimize the object code by exploiting the instruction level parallelism as much as possible. Usually this is done by code optimization such as unrolling, percolation scheduling[4], and software pipelining[5]. These optimization methods work extremely well on loops with simple data dependency.

But with loops with conditional branchs, with not enough instruction level parallelism, just applying optimization methods will not be enough effective. Actually in this case there is no way to get more instruction level parallelism, instead by speculatively executing the operations under the constrains of the conditional branch.

From the previous works [1][2][3], if the speculation are all done correctly it is reported that you can achieve large speedup in comparison without speculative execution. The problem is how to predict whether the conditional branch in loop would be taken or not at compile time. Recent compilers get these information by profiling the execution of the program, or guess it by using some heuristic in the compiler at compile time. This means which other method is taken, the guess are done completely statically at compile time. So they do not work effectively on conditional branches which its character differs completely when the input data of the program changes.

This paper will present a method to predict the behavior of the conditional branch inside the loop, and change the version of the loop code dynamically, which are optimized differently, using special designed branch prediction hardware. By the special designed branch prediction hardware, the processor can select the most appropriately optimized code to execute on runtime in each iteration, depending on the prediction done by the special branch prediction hardware.

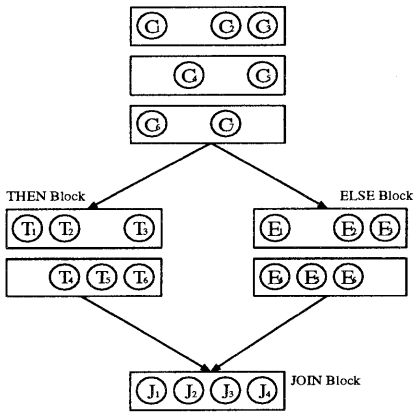
2 Compiler Supported Branch Prediction (CSBP)

This section will explain the basic idea of compiler supported branch prediction (CSBP). And also show how this CSBP method can be used in achieving more performance from the processor.

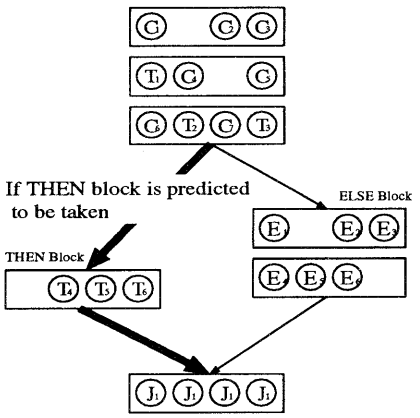
2.1 Basic Idea of CSBP

Figure 1 describes the main idea of CSBP. If there is a loop body that contains such code as Figure 1(a). The code can be compacted by moving up instruction from the THEN block, ELSE block and JOIN block to fill the resources of the processor. On compacting the code, if the branch possiblity is not known, compiler will first try to compact the code by moving up the operation from the JOIN block. This is because the JOIN block is always executed no matter whether the THEN block or ELSE block is taken. If the compiler could not find enough operation to move up from the JOIN block, then next it will start finding operation to move up from the THEN block and ELSE block to compact the loop body. But here is the problem, if there is no information about the branch possibility, or the branch possibility changes drastically depending on the input data, the compiler can not decide which path to move up operation from the THEN block or from the ELSE block. This means in the worst case the compiler will select the wrong path to optimize, and fail to utilize the resource by the operations in the correct path, which means it will be filled by operations which results will be never used and lose performance.

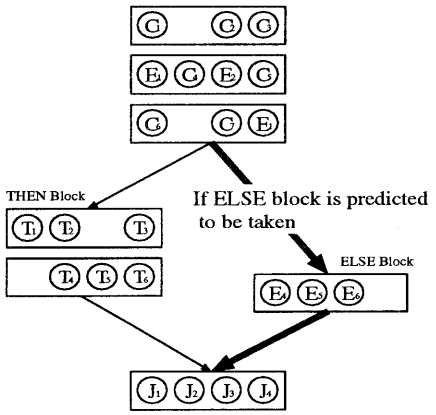
So if there is a method to select the most optimal code to run for each iteration, it may be possible to run the processor with no performance loss caused by executing miss speculated operations. The target of CSBP is to realize this code selection. For an example by preparing two versions of code from the same source code, the THEN block optimized code (figure 1(b)) and the ELSE block optimized code (figure 1(c)), CSBP method can make a good guess on selecting the most preferable code to execute based on runtime, by using similar method used in branch prediction for instruction fetch. To make CSBP easy to explain, we limited the number of the path to THEN block and ELSE block, but there is no need to limit the number of path CSBP can handle. CSBP method can also be applied on codes with more than two path inside the loop body. The implementation of CSBP will be discussed on section 3.



(a)Original Program



(b) Move up operation in THEN Block



(c)Move up operation in ELSE Block

Figure 1: Multi-version Loops

2.2 CSBP with two levels of speculation

CSBP can be used in every level of speculative execution. We categorize the speculation in the next two levels.

1. Non side-effect level: One level is moving up operations from the THEN and ELSE block which has no side-effect to the results, no matter either the THEN block or ELSE block is executed. For example, operations used to calculate address for load/store in THEN and ELSE block can be moved up over the branch operation for speculative execution with no side effects.

2. Side-effect level: The other level is moving up operations from the THEN and ELSE block which may cause side effect to the results, when the predicted path is not taken. But when the predicted path is taken, more performance can be achieved than the former level. Usually to avoid side effects hardware support such as shadow register[6] is needed for efficient execution. This is also called boosting.

CSBP can support both level of speculative execution effectively. Currently either of these two levels of speculative exection is used by statically predicting the branch direction, and speculating the operations of the predicted path. Combining the CSBP, will make it possible to speculate operations more effectively by predicting the branch direction dynamically.

3 Implementation of Compiler Supported Branch Prediction (CSBP)

CSBP can be implemented by adding two special machine instructions to the instruction set architecture. One is *Check Path (CP)* and the other is *Branch by Prediction (BP)*.

- *Check Path (CP)*: *CP* instruction is used for profiling the execution of the specific path. Machine instruction *CP* will take two operands, first one is for specifying the register number

used for writting the results of branch prediction with the execution history of the execution path, second operand is used for specifying the branch prediction scheme used with CSBP. The limit of number of registers *CP* can specify, will limit the number of path CSBP method can handle.

- *Branch by Prediction (BP)*: *BP* instruction is a conditional branch instruction, which branch condition depends on the branch prediction. Branch prediction are done from the branch history written by the *CP* instruction in the register specified as an operand. Machine instruction *BP* will take two operands, first one is register number to read the branch history, and the second operand specifies the target address to branch.

Figure 2 is an example how these two instructions are used in the loop body by the CSBP method. Figure 2 contains two versions of code compiled from the same source code. One version of code (a) is loop body optimized as the THEN block is predictied taken, other version of code (c) is loop body optimized as the ELSE block is predicted taken. When a simple one bit branch prediction scheme[1] is used for branch prediction, the execution of the loop will be done as the following.

The execution of the loops will start the execution of code (a) from label THEN_OPT. Depending on whether the THEN block or ELSE block of code (a) is executed, the result of branch prediction will be updated. The detection of whether THEN block or ELSE block is taken is done by checking whether *CP* is executed before *BP*. Because we are assuming a simple one bit branch prediction scheme, if the THEN block is taken *BP* (instruction (b)) will branch to label THEN_OPT to start the next iteration with the THEN block optimized code. If the ELSE block is taken *BP* (instruction (b)) will not branch and start from label ELSE_OPT to start the next iteration with the ELSE block optimized code. ELSE block optimized code (c) works as same as THEN block optimized code (a). If the THEN block is taken *BP* (instruction (d)) will branch to label THEN_OPT to start the next iteration with the THEN block optimized code. If the ELSE block is taken *BP* will not branch, instead branch instruction (e) will branch to label ELSE_OPT to start the next iteration with the ELSE block optimized code.

[1]The result of branch prediction always follows the results of previous iteration.

4 Experiments

4.1 Benchmark program

To show how effectively CSBP can work, we have done some simple experiments with a small simple program called RTFLSP. The core loop of RTFLSP is shown at Figure 3. RTFLSP is a program to find the root of a function fx(x) known to lie between the number xl and xh, by using the false position method. From the character of the program, whether the THEN block or ELSE block is taken will change depending on the inital value of xl and xh.

4.2 Target architecture

VLIW processor architecture was used for the experiment. The target VLIW processor architecture contains 128 registers with the following function units that can be used without any conflicts in resource, two arthimetic logic unit, two shifter unit, two floating point arthimetic logic unit, two floating point multiply unit, two floating point divide unit, and two memory access unit. To make the experiment simple we assumed all of these function unit have latency of one cycle.

4.3 Object Code generation

The object code of RTFLSP was created by adding *BP* and *CP* instructions to the code generated by the Percolation Scheduling VLIW compiler developed in UCI[2]. The source code of RTFLSP was compiled into two version of code by changing the weight of path. One version of code was THEN block optimized, and the other version was ELSE block optimized. Since the compiler used for generating the code does not currently support the special hardware for speculative execution with side effect, the level of speculative execution is now restricted on non side-effect level. The generated two versions of code are shown in Figure 4. Each rectangle box numbered with a hex number

[2]http://www.ics.uci.edu/~ snovack/overview/project_description.html

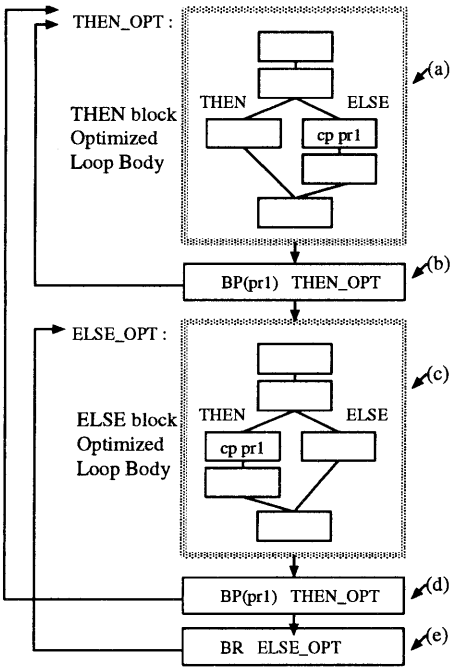


Figure 2: Object code of CSBP

```
for (j = 1; j <= MAXIT; j++){
        rtf = xl + dx * fl / (fl - fh);
        f = fx(rtf);
        if (f < 0.0){
                del = xl -rtf;
                xl = rtf;
                fl = f;
        }else{
                del = xh - rtf;
                xh = rtf;
                fh = f;
        }
        dx = xh - xl;
        if ((fabs(del) < xacc) ||
            (f == 0.0)              ){
                return(rtf);
        }
}
```

Figure 3: Source code of RTFLSP

shown in Figure 4 represents one VLIW instruction, which contains maximum twelve operation that can be executed simultaneously. By counting the number of the box in the path, you can estimate the number of cycles needed to executed one iterations.

4.4 Results

From Figure 4, we can see the following results.

1. If the THEN block is taken when running an THEN block optimized version, 10 cycles are need to execute one iteration.

2. If the ELSE block is taken when running an THEN block optimized version, 12 cycles are need to execute one iteration.

3. If the THEN block is taken when running an ELSE block optimized version, 12 cycles are need to execute one iteration.

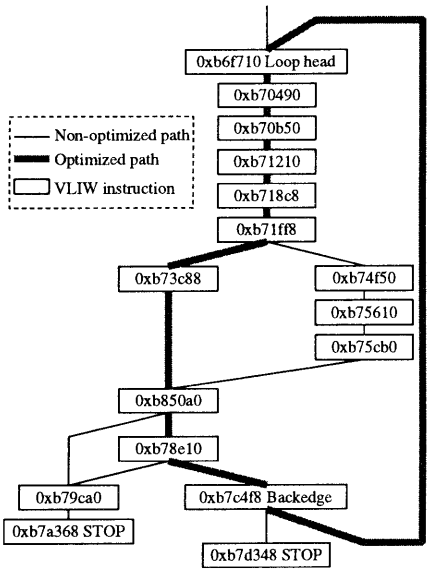
4. If the ELSE block is taken when running an ELSE block optimized version, 10 cycles are need to execute one iteration.

Since the conditional branch used in RTFLSP code chooses its direction of branch by the inital data and does not change its branch direction later, the execution time of the program can be estimated very easily. From the numbers achieved from Figure 4, it can be said that CSBP can achieve 0 – 20% inprovement in preformance, compared to the static profile based compilation method, which are usually used.

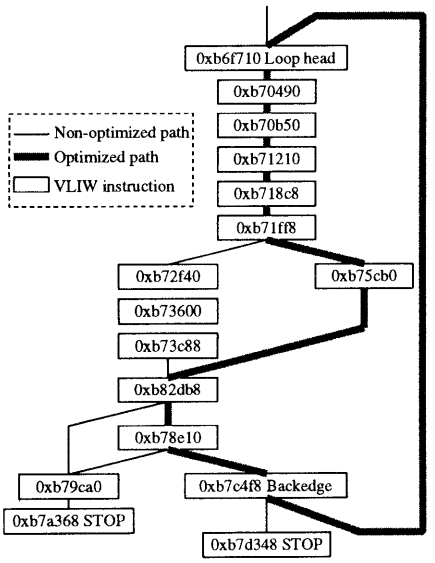
Our current experiment is only based on non side-effect level speculative execution. By combining special hardware support for side-effect level speculative execution, we believe the CSBP method will be more effective.

5 Conclusion

In this paper, we have shown the basic idea of Compiler Supported Branch Prediction (CSBP), and described the instruction set needed to be added to the processor to implement CSBP. To approve the idea of CSBP we did a simple estimation on non side-effect level speculative execution and achieved 20% improvement compared to the



(a) THEN block optimized version



(b) ELSE block optimized version

Figure 4: The generated two versions of code (RT-FLSP)

static profile based compilation method, which are usually used.

As more aggressively the speculative is done, it will be more important to speculate the correct operations. We believe the CSBP method will achieve more performance on higher level of speculative execution, such as side-effect level speculative execution described in section 2.2. For futher research we are planing to estimate the performance of CSBP on side-effect level speculative execution.

6 Acknowledgements

The authors would like to thank Steve Novack in UCI for the usefull discussion, and for the help on using the VLIW compiler.

References

[1] E. M. Riseman and C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Trans. Computer, 21, 12,* 1972.

[2] A. Nicolau and J. A. Fisher, "Measuruing the Parallelism available for Very Long Instruction Word Architecture," *IEEE Trans. Computer, 33, 11,* 1984.

[3] M. S. Lam and R. P. Wilson, "Limits of Control Flow Parallelism," *Proc. of the Ann. Symp. on Computer Architecture,* 1992.

[4] A. Nicolau, "Uniform parallelism exploitation in ordinary programs," *Proc. of the Int. Conf. on Parallel Processing,* 1985.

[5] Monica S. Lam, A Systolic Array Optimizing Compiler, Kluwer Academic Publishers, 'pp. 83–145, 1989.

[6] M. D. Smith, M. S. Lam and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," *Proc. of Ann. Symp. on Computer Architecture,* 1990.