Recovery Scheme to Reduce Latency of Miss-Prediction for Superscalar Processor using L1 Recovery Cache

Ye JiongYao and Watanabe Takahiro
Graduate School of Information, Production and Systems, Waseda University
Tel: +81-93-692-5371, Fax: +81-93-692-5371
E-mail: yejy_asgard@suou.waseda.jp
E-mail: watt@waseda.jp
Keywords: recovery, superscalar, branch miss-prediction

Abstract: A branch prediction is indispensable to modern superscalar processors for high performance. Although it has great possibility to improve performance, the advantage may be lost due to miss-prediction. To reduce such a branch miss-prediction penalty, finer recovery mechanisms are needed. One of those mechanisms is a RcB (recovery buffer), which preserves instructions to be restarted when miss-prediction occurs. But RcB cannot recover instructions issued out of order for a superscalar processor.

This paper proposes a L1 recovery cache embedded in a superscalar processing, RcC for short, which overcomes weakness of RcB and can recover instructions issued out of order so that recovery penalty is reduced. Our proposed L1 RcC scheme can work supplementing the conventional 1-bit dynamic branch predictor used in a superscalar processor, so that miss-prediction penalty can be effectively reduced.

1. Introduction

Branch hazards result in increasing CPI, especially as pipelines become longer. This drawback appears more obviously when a processor issues multiple instructions per cycle. Therefore an effective branch predictor has been researched to reduce branch hazards. But, even if a predictor scheme is the best, miss-prediction cannot be avoided as far as a speculative execution technology is used. Furthermore, a predictor scheme which is effective for some program may not be good for another. So, a finer and smart recovery mechanism is needed to reduce recovery penalty. Currently there are two approaches to reduce the penalty: one is incremental rolling-back to in-order state by waiting until the miss-predicted branch reaches the head of the reorder buffer, and another is utilizing check-pointing at branches for faster recovery[4][5]. But, the former method stalls the pipeline for a significant number of cycles and the latter is costly to design due to a complex structure. In order to overcome such difficulties, we proposed an improved recovery scheme [2], which allows the miss-predicted instructions stored in a buffer called RcB (recovery buffer) to be used at the next miss-prediction. It can also reduce latency of miss-prediction, however it can only store the instructions in order and recover in order. Therefore the scheme is not applied to a superscalar processor where the instructions may be issued out of order.

This paper proposes a complement of recovery buffer to adapt to a superscalar processor employing a L1 Recovery Cache (RcC for short), which can decode and dispatch in the cache, and it allows the instructions to be recovered out of order.

Furthermore, the outcome of speculatively-executed instructions can be stored in L1 RcC, even if the instructions are miss-predicted. At the next recovery step, this outcome in RcC can be directly read out and used if the relative source registers do not change.

Figure 1 shows an example of L1 RcC in a superscalar processor [3] that based on the DLX processor [1].

The features are

Two instructions per cycle and 7 phase pipeline

Branch-Target buffer (1-bit predictor)

Reorder-Buffer to commit instructions in program order (may be replace by L1 RcC)

A multiple reservation-station followed by four execution units

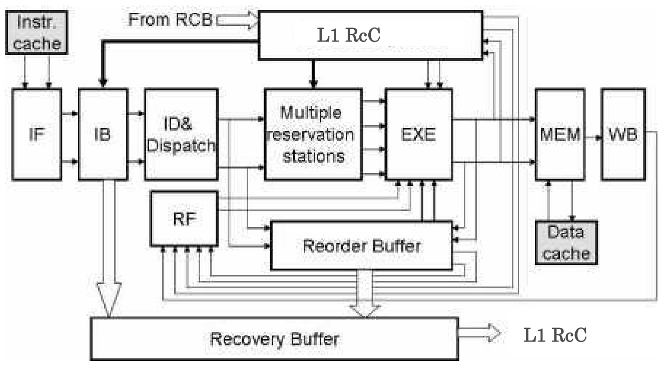

Figure 1 Two-issue Superscalar Processor with L1 recovery cache

2. preparation work

We have proposed a recovery mechanism using Recovery Buffer, , which allow processor recovering correct path without fetching instructions from instruction cache, trying to execute the instructions even if it is in miss-prediction path, solving recovery of multi pending in-flight branch. First, we further define two terminologies:

1. *Recovery Block* (RB): a set of all the instructions, which is in recovery path according to sequential program order. The maximum size of recovery block is as the same as reorder buffer, but commonly we can set a small size by depth of pipeline.
2. *Predictor path* that is instruction queue that predictor by a pending in-flight branch.
3. *Miss-predictor path* that is predictor instruction queue when relative branch predictor is missing.
4. *Recovery path* that is the candidate instruction queue for

the relative branch.

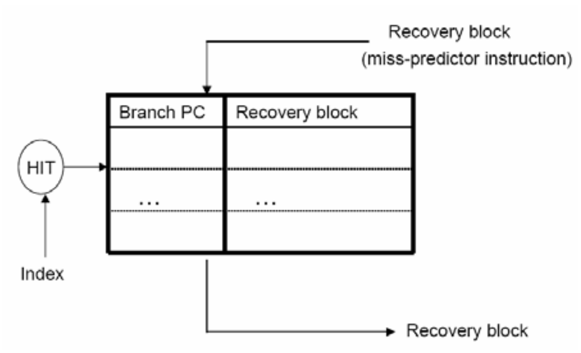

Figure.11 Recovery Buffer Operation

The major task of RcB is stored the miss-predictor instructions. Recovery block as the base unit to store instructions. Figure.11 shows the operation of the recovery buffer.

When a branch instruction is retired from the base pipeline, the relative recovery buffer can be copy into recovery buffer according to this branch instruction next predictor value. For example, if the branch instruction next predictor value is 1 (means taken), the set of instructions that not taken can be stored into RcB.

When a branch instruction is fetched by base pipeline, the relative recovery blocks are used minor recovery path to transmit from RcB to L1 RcC if Branch PC saving in RcB indexed by low-order bits of branch address, tag holds high-order bits.

3. A proposed recovery mechanism

Figure.2 shows the placement of the L1 RcC + RcB in the customary pipeline [1]. Every cycle, instruction can be issued from Decode/Rename pipelines, from the L1 RcC or from both structures to the execution pipelines

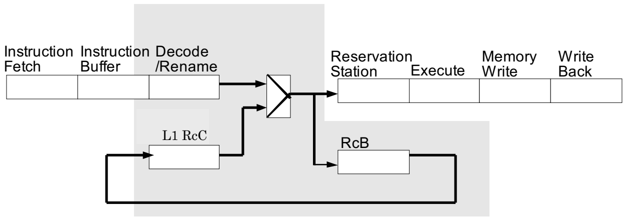

Figure 2 Recovery scheme by RcB and L1 RcC in a processor pipeline

When a branch instruction is issued to branch execution unit, relative recovery block can map into L1 recovery cache from RcB, and complete renaming in this cache. Preserving instructions in L1 Recovery Cache in sequential program order can ensure correct rename action, then issue the recovered instructions into reservation stations to wait source operands being completed.

Upon a recovery action, the L1 RcC replaces the predictor path (holding in reorder buffer) as recovery path (holding in L1 RcC) to issue the instructions into dispatch pipeline out of order. When all of instructions in L1 RcC are completed, L1 RcC commits the completed instructions in order, and returns the pipeline to base pipeline. The speculative executed result can be bypass to other instructions when needed if the relative resource is not change.

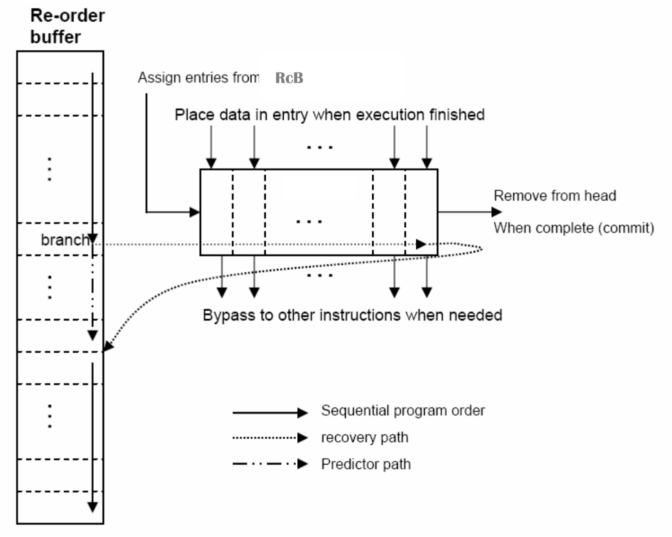

Figure 3 L1 recovery cache Operation

Figure 3 illustrates operation of L1 recovery cache. L1 RcC is easiest to think of it as parallel-input and serial-output mode storage, implemented in hardware as a circular cache which head and tail pointers. As a branch instruction is executed, L1 RcC assign entries for relative recovery block in order, then the instructions issue into reservation stations if relative branch predictor is miss. As instructions complete execution, their result values are inserted into their previously assigned entry, wherever it may happen to be in the L1 RcC. At the time an instruction reaches the head of the L1 RcC, if it has completed execution, its entry is removed from the cache and its result value is placed in the register file. An incomplete instruction blocks at the head of the L1 RcC until its value arrives. Of course, L1 RcC must be capable of putting new entries into the cache and taking them out more than one at a time.

Figure.4 shows the rename process applied to the same add r3, r3, 4 instructions. At the time the instruction is ready for dispatch, the values for r1 through r2 reside in the register file. However, the value for r3 resides (or will reside) in L1 RcC entry 6 until that L1 RcC is committed and the value can be written into the register file. Consequently, as part of the renaming process, the source register r3 is replaced with the L1 RcC entry 6 (rcc6). The add instruction is allocated the L1 RcC entry at the tail, entry number 8 (rcc8). This L1 RcC number is then recorded in mapping table for instructions that use the result of the add.

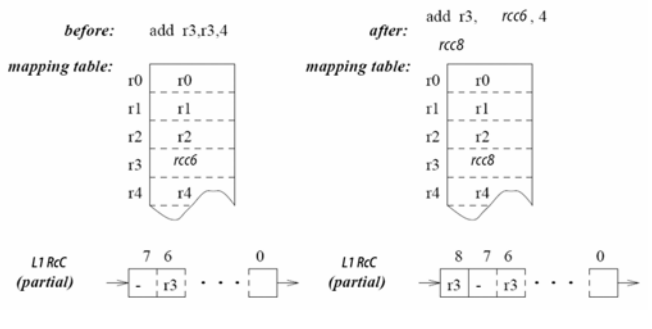

Figure 4 example of renaming with L1 RcC

4. Structure of L1 Recovery Cache

Figure 5 shows a structure of L1 RcC. It consists of many fields for entries of data, that is, 'Instruction field' to keep a recovered instruction, 'Data field' to keep speculative outcome. 'Commit index (Co)' to allow the outcome to commit, 'Busy (Bu)' to indicate whether this outcome of instruction is valid or not, 'Ready (Re)' to decide whether the outcome that keeps in Data can be speculatively used by related instructions, 'Source registers SR1 and SR2' to hold pointers to where the data can be found, and 'Valid fields V1 and V2' to indicate whether the source value is ready or not.

Instruction						
SR1	V1	SR2	V2	Bu	Co	Re
Data						

Figure 5 Structure of L1 RcC

5. Conclusions Simulation methodology

We simulate our proposed customized the new recovery mechanism on a simulated, generic processor, loosely modeled on a DLX processor simulation tools derived from the MIPS-IV ISA [6]. The simulation named DLXsim is an interactive program that loads DLX assembly programs and simulates the operation of a DLX computer on those programs, allowing both single-stepping and continuous execution through the DLX code. DLXsim also provides the user with commands to set breakpoints, view and modify memory and registers, and print statistics on the execution of the program allowing the user to collect various information on the run-time properties of a program. We expect that a major use for this tool will be in association with future CS 252 classes to aid in the understanding of this instruction set.

We decided that since the MIPS instruction set has many similarities with DLX, and a good MIPS simulator already exists, it would be a better use of our time to modify that simulator to handle the DLX description. This simulator was built on top of the Tcl interface, providing a programming type environment for the user as well. The main problem we encountered when rewriting the simulator was that there are a couple of fundamental differences between the DLX and MIPS architectures. Following is a list of the main differences we identified between the two architectures.

- ➢ In MIPS, branch and jump offsets are stored as the number of words, where DLX stores the number of bytes. This has the effect of allowing jumps on MIPS to go four times as far.
- ➢ MIPS jumps have a non-obvious approach to determining the destination address: the bits in the offset part of the instruction simply replace the lower bits in the program counter. DLX chooses a more conventional approach in that the offset is sign extended, and then added to the program counter.
- ➢ In the MIPS architecture, conditional branches are based on the result of a comparison between any two registers. DLX has only two main conditional branch operations which branch on whether a register is zero or non-zero.
- ➢ DLX provides load interlocks, while the MIPS 2000 does not.
- ➢ MIPS 2000 provides instructions for unaligned accesses to memory, while DLX does not.
- ➢ The result of a MIPS multiply or divide ends up in two special registers (HI and LO) allowing 64 bit results; the result of a DLX multiply is placed in the chosen general purpose register, and must therefore fit into 32 bits.

Because of the large number of similarities between DLX and MIPS, we based our opcodes on those used by the MIPS machine (where MIPS had equivalent instructions). Where DLX had instructions with no MIPS equivalent, we grouped such similar DLX instructions and assigned to them blocks of unused opcodes. Base on this DLX simulator, we still should modify the architecture of DLX simulator. We use 2K entry 2 level PHT branch predictor and simulate 4K entry recovery-buffer. Before we simulate the performance of recovery-buffer, let us estimate the performance by the following formula:

$$f = (p_{mis} - p_{lop}) / p_{mis}$$

f : Performance of recovery mechanism
p_{mis} : Miss-predictor of program
p_{lop} : A piece of branch instruction may repeat executing within the program is running. So it is means that how many time a branch instruction repeat except the first time.

The test benchmark:

- ● nested loop
  - ➢ description: a simple nested loop program
  - ➢ test the improved prediction accuracy.
- ● fibonacc
  - ➢ description: a simple recursive program
  - ➢ test the effect of the recursive program.
- ● permute
  - ➢ description: a more complicated recursive program
  - ➢ test the effect of more complicated recursive program.

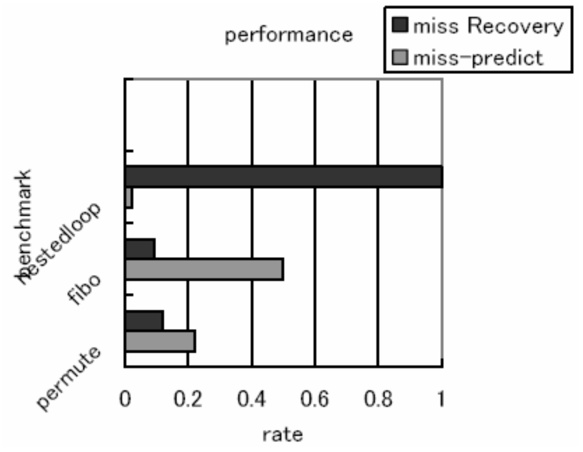

Figure 6 Performance of the Recovery Mechanism

Figure 6, we can fine that this recovery technology is very fit to recursive program. Alone with the depth of recursive become large, the performance of processor can be more improved by using the recovery-buffer.

6. Conclusions

This paper proposes a new recovery mechanism using L1 RcC to deal with miss-predicted instructions in a superscalar processor. It can resolve a difficulty in a traditional mechanism where miss-prediction penalty is not reduced for instructions issued out of order. The proposed L1 RcC can also improve performance of a superscalar processor for speculatively- executed instructions.

Currently, we are implementing this scheme on FPGA, and then we will investigate the actual performance experimentally to verify the effectiveness of the proposed mechanism and the superiority to a conventional one.

References

[1] John L. Hennessy and David A. Patterson, "Computer Architecture: A Quantitative Approach, Third Edition", Morgan Kaufmann Publishers, 3rd edition, 2003.

[2] Ye JiongYao and Watanabe Takahiro , "Performance Improvement for Branch Prediction Processing", 2005 IEICE General Conference D-6-3, March 2005.

[3]John Paul Shen and Mikko H. Lipasti, "Modern Processor Design fundamentals of Superscalar Processors", McGraw-Hill Publishing 2004.08.

[4]H. Akkary, R. Rajwar, and S. T. Srinivasan, "An analysis of a resource efficient checkpoint architecture", ACM Transactions on Architecture and Code Optimization, Volume 1, pp.418–444, December 2004.

[5] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry, "Checkpointed early resource recycling in out-of-order Microprocessors", Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35), pp.3–14, Istanbul, Turkey, November 2002.

[6] Charles Price. MIPS IV Instruction Set, revision 3.1. MIPS Technologies, Inc., Mountain View, CA, January 1995.