RC-002

Reducing Branch Misprediction Penalty in Superscalar Microprocessors by Recovering Critical Misprediction

叶 炯耀 † 万 宇 † 董 宜平 † 鮑 治国 † 渡邊 孝博 †
Ye Jiongyao Wan Yu Dong Yiping Bao zhiguo Watanabe takahiro

1. Abstract

In modern superscalar processor, branch misprediction penalty becomes a critical factor in overall processor performance, especially in deeply pipelined processors. The branch misprediction penalties include branch resolution time and refill the pipeline. A large number of aggressive schemes (e.g. checkpoint scheme) are widely used in most of current approaches to reduce the branch resolution time. However, current recovery mechanisms still implicitly reduce the Instruction Per Cycle (IPC) because the mispredicted instructions saved in the front-end stages must be flushed, and then the instructions from correct path are restarted from fetch stage.

In this paper, we propose a recovery mechanism, called Recovery Critical Misprediction (RCM), to reduce the branch misprediction penalty due to re-fill and flush. The mechanism uses a Simplicity Trace Cache (STC) to trace mispredicted instructions that are enough critical, and selectively forks a second path from STC following a conditional branch instruction. Upon a misprediction, the processor can immediately starts issuing correct instructions from the alternate path. Experimental results employing SPECint 2000 benchmark show that, using a processor with RCM, IPC value is significantly improved by 10.7% on average compared with a conventional processor without RCM.

2. Introduction

A deeper pipeline is widely used to reduce processor cycle time for higher performance in the modern processors, it causes another performance problem for branch misprediction. The research proposed by E. Sprangle indicated that branch mispredictions are the single largest contributor to performance degradation in modern superscalar processors [1]. Two options exist to solve this problem: increasing prediction accuracy [19] [21] and speeding up the misprediction recovery process. Several branch prediction mechanisms have been proposed [2] [3] and used to alleviate the effect of branch penalty in processor performance. On the other hand, modern processors provide deeper pipeline (e.g. 14 stages in the IBM Power 4 [6] and 20 stages in the Pentium 4 [8]) to achieve a very high clock frequency. As a result, the performance bottleneck in advanced processor designs continually shifts toward the penalty due to the misprediction recovery.

A large number of aggressive recovery schemes have been widely used to reduce the mispredicted branch resolution time by reducing the time of state restoration [24]]25] [26] . It is because that branch misprediction recovery requires stalling the front-end of the processor to repair the architectural state. However, branch misprediction still implicitly reduces the Instructions Per Cycle (IPC) because the pipeline still must be flushed, and re-filled with instructions from the correct path after the state is resolved.

† Graduate School of Information
Productions and Systems, Waseda University

During the re-fill time, there is a zero-issue region where no instructions issue, which is approximately equal to the time it takes to re-fill the front-end pipeline (i.e., a number of clock cycles equal to the front-end pipeline length).

To resolve the problem, a more jacobinical approach is proposed to hide the re-fill time by executing multiple program paths simultaneously [7] [13] [20]. By increasing the hardware cost, the processor can fork a second path from both paths following a conditional branch instruction. Then, the instructions from wrong path are selectively flushed when the branch is solved. There are still three factors that severely degrade performance: 1) fetching from multiple path increases the burden of fetch stage. The port of fetch stage must be double to simultaneously fetch instructions from both paths following a conditional branch instruction. 2) Miss hit rate of instruction cache increases because the alternative path may not be in the cache. 3) Forking a branch made for non-critical dependences will not improve performance; even worse, if the current branch prediction is correct, the unused instructions from alternative path may severely degrade performance.

In this paper, we propose a new mechanism called Recovery Critical Misprediction (RCM) to aim at minimizing the branch misprediction penalty. The mechanism uses a Simplicity Trace Cache (STC) to trace a few decoded instructions that are enough criticality. Then during subsequent branch predictions, if STC is hit, the instructions from alternative path are selectively fed to the rename stage with predicted instructions at the same time, and immediately provided to execution unit when the misprediction occurred. Therefore, the processor does not need to start fetching the correct path from fetch stage. RCM makes the following contributions:

RCM reduce the misprediction penalty caused by re-fill and flush. A small simplicity cache is used to save the decoded instructions from the alternative path, and selectively uses these instructions according to the confidence mechanism of the branch:

1. RCM provides a critical mechanism to filter the instructions. The critical mechanism ensures the STC to select "good traces" for keeping, and avoid the non-critical branch forking to degrade performance.
2. RCM efficiently reduces the burden of fetch process that is big bottleneck in modern processor. The instructions from alternative path are directly provided by STC at rename stage, and recessively reduce the instructions cache miss rate due to fetching instructions from alternative path. (Although some dual fetch port techniques are used to resolve the above problem, it is difficult to apply it in embedded processor [13] [22] [23].)

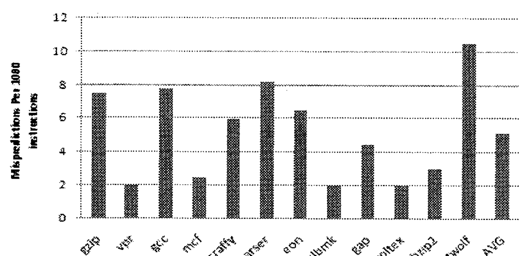

Figure 1 Branch Misprediction per 1000

Experiment results using SPECint2000 benchmark show that average IPC improvement is 10.7% compared with a processor without RCM.

The remainder of this paper is organized as follows. Section 3 presents the motivation of our proposal and related work. Section 4 analyzes the contributors to the branch misprediction penalty. Section 5 introduces the critical path prediction. The detailed design of RCM is described in section 6. Section 7 describes our experimental evaluation and discusses RCM performance. Finally, Section 8 summarizes the main conclusions of this work.

3. Background and Related Work

As the trend of utilizing deeper pipeline reduces processor cycle time from high performance, the branch mispredictions are a significant hinder to performance. The overall performance penalty due to branch mispredictions is the product of the branch misprediction rate and the branch misprediction recovery. A very large body of research has been targeted at improving branch prediction rate [2] [19] [21]. Currently, more and more researches begin to tend to reduce the branch misprediction penalty. There are two major ways of reducing the branch misprediction penalty. One way is speeding up misprediction solution time. For example, MIPS R10000 [4] using the retirement map technique, and Alpha 21264 [5] employing the Global Checkpoints to resolve misprediction within very few cycles. The other is that reducing the misprediction penalty due to re-fill and flush. Commonly, the processor executes multiple program paths simultaneously to hide the re-fill time (e.g. the IBM 3168 and 3033 mainframes could fetch instructions from both paths [28]).

Early researches propose dual fetch/decode mechanisms in a very simple pipelined processors [15] [23] to reduce the penalty due to misprediction. On the other hand, a special-purpose cache called Misprediction Recovery Cache is proposed to achieve the same purpose for an in-order CISC pipeline [29]. A more aggressive approach is proposed in [20], where a selective dual path execution (SDPE) allows executing instructions on both branch paths when there is a relatively high likelihood that the prediction will be wrong. All of those either need high hardware cost to support or are only suitable for simple pipeline and CISC.

In [7], a Dual path Instruction Processing (DPIP) fetches, decodes, and renames, but it does not execute instructions from the alternative path for low confidence predicted branch at the same time as the predicted path is being processed. This method reduces the re-fill penalty, and achieves a good trade-off between performance and complexity. Although DPIP employs the confidence mechanism to improve the usage of the instructions from alternative path, the performance would be decreased because the non-critical branch instructions with low-confidence are forked, and dual fetch mechanism increases the burden of fetch stage. Especially, an embedded processor cannot afford excessively complicated fetch architecture design.

4. Branch Misprediction Penalty Analysis

This section analyzes the branch misprediction penalty. The branch misprediction penalty is defined as the number of cycles lost due to a mispredicted branch. At first we have evaluated mispredictions per 1000 instructions for the SPECint 2000 benchmarks (details regarding the experimental setup are given in section 7). The results are shown in Figure 1. Based on the further analysis proposed by Eyerman [27], the sources of performance loss due to branch mispredictions are divided into five components:

1. *The frontend pipeline re-fill time*. It is the latency between the time that the branch misprediction is discovered and the time that the first instruction is fetched, decoded, renamed, and issued to instruction window. The pipeline re-fill time is a fixed value.
2. *Drain time*. This is the resolution time to drain incorrect instructions from the Reorder Buffer (ROB). The drain time strongly dependences on the program's inherent ILP. Programs with low ILP tend to have a large drain times, conversely, programs with high ILP tend to have a shorter drain times.
3. *Window-fill penalty*. This is the performance loss because there is a zero-issue region where no instructions issue until the instructions from correct path has been issued into instructions window.
4. *Non-unit latencies*. This is the performance loss due to the functional unit mix. The non-unit latencies are proportional to instruction execution latencies
5. *Short D-cache misses*. This is a factor easy to be neglected. An ideal L1/L2 D-cache is assumed. Actually, the branch misprediction penalty is affected by the miss rate of L1 or L2 cache.

Benefiting from above analysis, the main contributors of misprediction branch penalty are simply divided into two categories. One is inherent constraint of processor (e.g., frontend pipeline length, latency of each execution unit, latency of miss cache etc.). The other is average critical path, which is more complex. Branch mispredictions do not occur in isolation; they interact with other miss events. The penalty for a particular branch misprediction often depends on the preceding miss event (conversely, it can affect the next miss event). For example, the misprediction penalty is hidden under the long D-cache miss penalty if the mispredicted branch is not fed by the long D-cache miss. On the other hand, we could potentially speed up the program more by caching the instructions that are critical. Therefore, we would rather only optimize the branch, following which there are more the critical instructions.

Our proposal mainly targets at hiding the latency of re-fill and flush simultaneously, considering the effect of critical path. The key is what is critical for a branch misprediction recovery (described in section 5).

5. Critical Paths Mechanism

The RCM is efficient in terms of power and access time due to small size of STC employed in it, but suffers from low utilization of the memory space. Critical path mechanism is used to increase its effectiveness despite the limited size. Critical path mechanism tells processor which branch is enough critical to decide whether the branch needs to keep. Based on Brian Fields and Ras Bodik's work [9] on criticality, we can easily know which instruction is from critical path.

5.1 Defining Criticality

How to exactly define the critical path through a program depends very much on the context in which criticality is being used. A static critical prediction, which commonly employs compilers for improving instruction scheduling, is only involved in inherent program bottlenecks. The critical path using a static critical prediction is constant throughout execution, it can frequently change in a dynamically-scheduled context: when no micro-architectural events occur, the longest dataflow chain will be critical, but cache misses and branch mispredictions can elongate otherwise short dataflow chains, making them critical. In general, execution in a dynamic machine comprises a number of potentially-critical paths, their interplay being determined by events at runtime.

Though other studies have acknowledged the complexity of such interactions, Fields et al. were the first to tackle them directly and to characterize them precisely. The critical path prediction defines criticality is a function of a program's dynamic dataflow patterns and their interaction with the underlying micro-architecture. Our work focuses on this approach. Based on the critical paths and critical nodes, we further present the idea of Critical Trace Length.

5.2 Critical Paths and Critical Nodes

According to the research proposed by Brian Fields and Ras Bodikm, dependence graph model (shown in Figure 2) divide an instruction into several parts corresponding to the different stages that an instruction goes through in a processor pipeline. The dependence graph model is determined by how many and what stages are chosen in this division. For our purposes, the most basic model was used, which divides an instruction into three (entry into the out-of-order window (dispatch), execution at a functional unit (execute), and exit from the out-of-order window (commit)). Each part is then considered as a node in a graph; so each instruction consists of three nodes and a program is a graph with number of nodes equal to three times the number of instructions executed. Dependencies exist between different nodes. For instance, obviously all the commit nodes of the instructions will depend on the execute nodes of the same instructions, and the execute nodes will depend on the decode nodes. All the decode nodes and commit nodes will depend on the previous ones if we issue in order and have a reorder buffer. Furthermore, if there are data dependencies between the instructions, a decode node or a execute node may depend on a previous execute node.

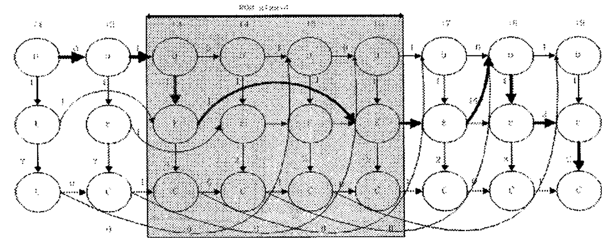

Figure 2 Dependence graph model. *The graph models a sequence of 9 consecutive instructions (from i1 to i9) in a program. The ROB is 4-entries. Instruction i7, a mispredicted branch, induces an E-D edge to instruction i8 to reflect the constraint that correct path instructions cannot be dispatched into the window until a misprediction is resolved. The critical path through this code sequence is highlighted with the thicker dependence edges. All edges are labeled with their latencies.*

With the dependency graph constructed, we can determine whether or not an edge is critical. An edge is defined to be non-critical if the overall run time stays the same while we reduce the weight on that edge. An edge is defined to be critical if it is not a non-critical edge. The critical path of a program is formed by following the edges that are critical.

A node is defined to be critical if it is part of the critical path. This notation is useful because we can use the technique of token passing along last arriving edges proposed in [1] to try to estimate the critical path. Obviously, an edge is not part of the critical path if it is not the last arriving edge of a node; we can decrease the weight of such edges and there would not be any performance gain because the node still has to stall until the last edge arrives. If we trace through all the last-arriving edges, we form an estimate of the actual critical path. Of course, this path may not be entirely correct, but this is a simple way to approximate the critical path. The estimation using last arriving edges can be relatively easily computed by hardware.

It is notion that branch mispredictions do not occur in isolation; they interact with other miss events. The penalty for a particular branch misprediction often depends on the preceding miss event. So whether the branch instruction is in critical path is the important parameter. If the branch instruction is in critical path, processor would fork the instructions following the branch instruction. Conversely, even if a non-critical branch is forked performance cannot be improved. For example, if the mispredicted branch is not fed by the long D-cache miss that the mispredicted branch immediately follows, and in this case the misprediction penalty is hidden under the long D-cache miss penalty that usually much larger than misprediction penalty.

5.3 Critical Trace Lengths

More critical-nodes after a branch mean more room for exploiting. So we tend to fork a second path that has more critical-nodes. The total number of critical node after a branch is defined as critical trace length. Figure 3 shows the average critical trace lengths for the SPECint 2000 benchmarks—details

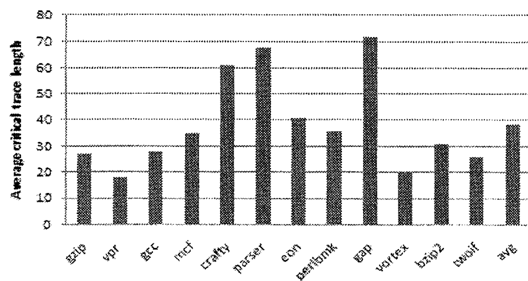

Figure 3 Mean Critical Trace Length

regarding the experimental setup are given in section 7. Based on the idea of criticality, the longer critical trace length has more room for exploiting. A long critical trace means that we could potentially speed up the program more by caching the instructions that are critical. Depending on how many instructions we store in our cache, it is possible to remove all the latency of the front-end pipeline for each instruction in the cache. Unfortunately it seems that many of the benchmarks exhibit extremely poor variance. So we still need to add threshold for filter of STC. The threshold value is measurable standard of critical degree. If the critical trace length is larger than the threshold value, it means the critical trace is sufficiently important to need to be saved in STC. In contrary, the critical trace cannot be saved in STC due to lacking criticality. Introducing the critical trace lengths can ensure the usage of STC.

5.4 Computing Critical Trace Lengths

Using the lengths of critical trace in any structure in an actual microprocessor would require an efficient hardware structure to compute the lengths of these chains. A first requirement is that the base architecture must include a criticality predictor, such as the one described in [9] based on token passing along last-arriving edges.

Once the criticality for each node can be predicted, there is a counter for each branch, at most one counter per Reorder-Buffer entry, which is incremented on each instruction that is critical nodes. The information in this counter must then be retained somewhere. Section 6.2 describes how we decided to retain the information in these counters.

6. Misprediction Recovery Cache

In this section, we introduce the RCM. At first, recovery trace is a critical trace saved in STC. RCM uses STC that is similar to traditional trace cache [10]. STC will be discussed in section 6.1.The operation of RCM will be explained in section 6.2. Recovery policy is discussed in section 6.3.

6.1 STC

We propose using a small simplicity trace cache with decoded instructions to reduce the branch misprediction penalty. Figure 4 shows that an STC is added into a basic out-of-order pipeline. Trace buffer takes input from the Decode stage of the processor and keeps a buffer of the current trace. This buffer stops taking input, and computes the critical trace length according to critical path prediction when it becomes full, or when its data is written into the trace filter. The length of critical trace is compared with threshold value to decide whether the critical trace is enough critical to be written into STC at the trace filter. The branch misprediction also triggers a lookup in the STC to see whether the recovery path of current misprediction is in the cache. If it is (meanwhile, it is low-confidence), then the trace is used as the input to the Rename stage by using distribution approach that the instructions are catch from the prediction path and the recovery path according to time interleaving. In essence, STC is a simplified trace cache. Most design issues of trace cache [10] [11] can be utilized by the STC. Two points different from a traditional trace cache should be paid attention: 1) *Adding critical evaluation mechanism*: STC latches instructions from decode stage, instead of fetch stage, and uses trace buffer that employs critical mechanism to compute critical trace lengths, and uses trace filter to catch the critical trace with enough critical. Benefiting from critical path analysis and recovery policy, STC does not need to consider complex state detection and management for multiple branches or other speculation techniques (discussed in section 6.3), so STC architecture is simple. 2) *Adding branch confidence mechanism*: the branch confidence mechanism prevents the high-confidence recovery trace from entering pipeline, thus increases the ILP of front-end pipeline, which will be discussed in 6.3.2.

6.2 RCM Operation

RCM allows two paths of branch entering the pipeline (rename stage and instruction window) simultaneously. But the instructions from recovery path are not executed until a prediction miss, and the pipeline only need to discard an invalid path after the branch is resolved. The issue windows cannot be drained because both paths following a conditional branch have been fetched. In essence, the approach converts control dependencies into data dependencies to reduce the misprediction penalty. If the STC is hit, and the current branch is low-confidence, the decoded instructions from the both path would be renamed and dispatched simultaneous, but considering power and efficiency, the instructions from recovery path are not executed until the branch is resolved. If the prediction is correct, the instructions from recovery path must be discarded. Conversely, predicted path must be discarded, and the recovery path holding in issue window will be issued into execution unit. It is noted that the false data dependencies are introduced due to converting control dependencies into data dependencies.

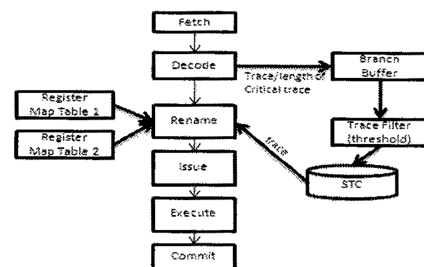

Figure 4 MRC Architecture

6.3 Recovery policy

Ideally, we would like to create the recovery path exactly at every branch, so that all branch mispredictions are avoided. Actually, three factors prevent performance improving according to this approach: 1) excessive instructions from recovery path reduce the ILP of the front-end pipeline. 2) Introducing excessive recovery path may make the hardware very complex and difficult to rename. 3) Forking the non-critical trace would not help performance improvement.

The critical path mechanism discussed in section 5 is used to tradeoff the trace criticality to help improving the usage of RCM. Then, we introduce other mechanism to improve the performance of RCM.

6.3.1 Branch Forking Policies

Delayed Forked Branches (DFB) scheme is used to reduce the cost of hardware [7]. DFB will prevent the second branch from forking a new path. The idea is to save the processor current states when the second low confidence branch is encountered. After the resolution of the current forked branch, the delayed branch can be forked using the previously saved state. To simplify the mechanism, only the first low-confidence branch fetched by the predicted path can be delayed. For this purpose, RCM also uses two Register Map Table/Free-list structures for the main path and other two for the alternative path, as Figure 4 shows. Based on the branch forking policies and critical mechanism, RCM truncates the recovery path when it encounter a branch that is both low-confidence and non-critical.

6.3.2 Branch Confidence Mechanism

We propose to selectively use recovery path to avoid excessive invalid instructions entering pipeline. In reality, we would always like the branch prediction is correct, so we limit the high-confidence branches from recovery path to create the checkpoint and recovery path. It is noted that RMC masks the low-confidence branches which related recovery path has be catch from STC to create checkpoint because the low-confidence branches have created recovery path.

A branch confidence mechanism sorts conditional branch predictions into low and high confidence sets based on previous predictability. Branch confidence mechanisms were already studied in depth in [12]. Here we exploit a simple effective confidence mechanism. The confidence mechanism consists of a table of resetting 4-bit counters, and is indexed by the XOR of a Brach PC and a global branch history register (GBHR). Correct predictions increment the counter and a misprediction resets the counter to zero. For checkpoint, a value of 15 signals high confidence which the remaining values signal low confidence. For RMC, a value of 3 signals high confidence which the remaining values signal low confidence.

6.3.3 Renaming Mechanism

RCM allows instructions from both paths to coexist in the Rename stage and instruction window. RCM duplicates the rename map table for shadowing current state, which is already done in some practical processor such as the MIPS R10000 using decoupled architecture [22]. This method stores a shadow copy of the register map as it exists when a conditional branch is predicted. Rolling back to the branch in the case of a misprediction involves replacing the current register map with the appropriate shadow map. After a forked branch is resolved, the register map for the incorrect path can be discarded. The register map for the correct path must be placed into the current register map for the predicted path, which is used when only one path is being executed. RCM also uses two register maps, one for each path. The reorder buffer (ROB) and the load/store queue structures are also duplicated. When a second path (i.e. recovery path) is forked, the current register map is copied into the forked register map. Thus, the maps used for each path are the same at the point of the branch fork. As instructions are renamed on each path, different physical registers are mapped to the instructions on each path, and the separate maps are used. It ensures fast state recovery.

7. Experimental Results

7.1 Simulation Methodology

The performance numbers presented below are based on an extended version of the sim-outorder simulator from the SimpleScalar tools set 3.0 [14] that was augmented with a detailed model of the trace cache that includes recovery, along with the simulation of the proposed RCM mechanism and with a critical path predictor using the configuration same as research [9] to achieve accurate prediction rate. The structure of the traces within the RCM is similar to other works [10]. Based on critical mechanism and confidence scheme, a trace is terminated if it reaches the first critical branch node which low-confidence. Each instruction in STC takes up 8 bytes since decoding instructions expands them into a less dense encoding, more conducive to use by the processor's data path. An aggressive prediction [19] is used in our simulation to verify the performance of RCM. Table 1 summarizes the base parameters of a basic processor and configuration of critical predictor.

All the SPEC CPU2000 integer benchmarks were used. All SPEC applications use the reference inputs. In order to reduce simulation time, we used the Simpoint [16] to choose representative samples of over 300 million instructions [17]. We compiled the SPECint CPU2000 benchmarks for the Alpha 21264/Unix using SPEC Peak compiler and link.

7.2 STC Design Space Exploration

Table 1 Configuration of the simulation

Fetch engine	Up to 8 instr./cycle. 128. Non-blocking I-Cache
L1 D/ L1 I Cache	64KB,32Byte/line,4-way set-associative, 2 cycles
L2 unified cache	512KB, 32Byte line, 4-way set-associative, 10 cycles
Main Memory	200 cycles
Branch prediction	8K-entry gshare predictor and 8K-entry bimodal, 16K selector. 2 branches per cycle
Issue/Decode/Commit	Any 8 instr./cycle
Scheduler	1K-entry size LSQ
Front-end Recovery	Fetch + Decode + Rename = 4 cycles
Function unit and latency (total/issue)	4 Int ALU(1/1), 1 Int Mult(2/1)/Div(20/19), 4 memory(1/1)
Critical path predictor	12Kb (16K entries * 6 bit hysteresis). 1024 dynamic instructions for token propagation distance. 8 Tokens in flight simultaneously.

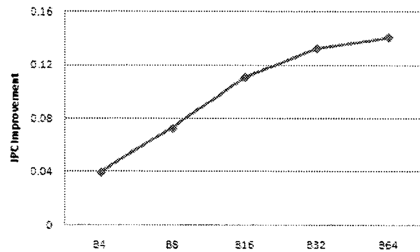

Figure 5 The improvement of the IPC with the different trace

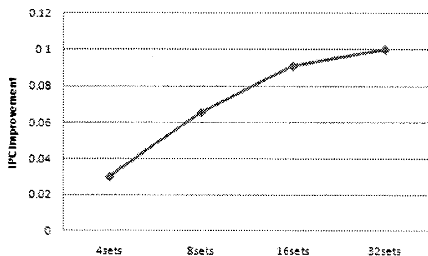

Figure 6 The improvement of the IPC with the different trace entries

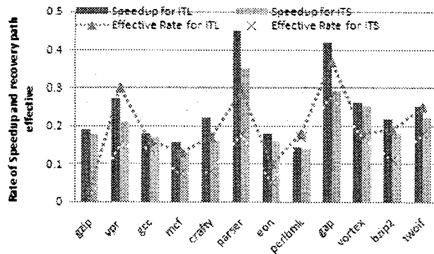

Figure 7 Comparing the performance of ITL and ITE

This section explores the design space of the STC. The basic STC is that the number of trace entry is 16, and trace length is specified to 32. The total size of STC is 4KB. The trace replacement uses Least Recently Used (LRU). We attempt to systematically vary the design search space for design via a series of experiments and analyses. We simulated the trace cache by reducing the branch misprediction penalty linearly with the number of critical trace length, so we only focus on some parameter of STC that is major effect for performance improvement: including of cache size, critical trace length and trace utilization.

7.2.1 Cache Size

To further increase the performance, the size of STC has to be increased. There are two ways to increase the size of STC. One is Increasing Trace Length (ITL). The other is Increasing Trace Entry (ITE).

ITL means that more instructions can be recovered after a misprediction. Figure 5 shows the improvement of the IPC with different trace length (For example, B32 means there are 32 instructions in a recovery trace). Observing Figure 5, trace length increases after certain degree, the performance enhancement becomes slow. This is because that the overlengthy recovery path is very rare. The recovery path can be truncated by the critical branch with low-confidence.

Otherwise, ITE means how many recovery paths can be written into the STC. Figure 6 shows the improvement of the IPC with the different trace sets (16sets means 16 trace entries can be kept in STC). In the same condition, the ITL more easily lead to waste of cache size than the ITE. But more the number of trace need more tag bit, practical performance comparison of the two methods are shown in Figure 7. By comparing the two performance index of STC (including speedup, and effective rate of recovery path), we can find that the ITL is better than the ITE on each index.

7.2.2 Critical Trace Length

Improving the usage of STC is a major purpose of our research, and the threshold value is the key factor for the performance of STC. The Threshold Value (TV) of trace filter determines whether the recovery path has enough criticality to be kept into STC. Commonly, long critical trace is better since there is more room for exploiting. The performance improvement from decreasing critical path latencies is much larger than from decreasing non-critical latencies. Any non-critical instructions may not benefit from early scheduler. On the other hand, excessive TV is also not appropriate. The trace paths are excessively limited by large threshold because the trace length must larger than TV. Figure 8 shows the performance using the various thresholds for per trace length. It is very obvious that the IPC cannot be improved if the trace length smaller than TV, and achieves the best performance near where the trace length is 2 times of TV. Then, the performance is diminished because the low-critical paths are imported.

7.2.3 Trace Utilization Rate

The Trace Utilization Rate (TUR) is defined to be the number of times the system finds the trace in the trace cache per a trace build. It is note this definition does not require that the traces are unique; i.e., if a trace is replaced and built again, we count it as two different traces. Also, the length of the trace does not affect the utilization of the trace. In figure 9 the trace utilization breakdown is presented for 8-entries traces and 16-traces trace caches respectively. In both configurations the majority of traces that are written into the trace cache are not used prior their eviction from it (TUR=0). Moreover, only 10% of the writes results with more than 2 hits (TUR>2) for the 8-traces trace cache and 20% for the 16-traces trace cache. We propose the STC that a trace can contain up to 32 instructions, 16 entries can

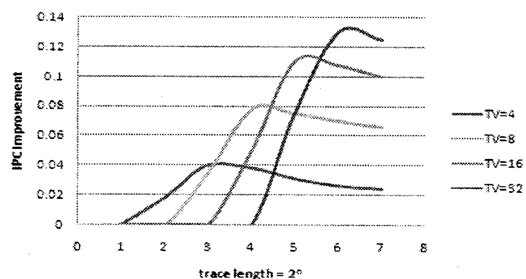

Figure 8 Various Threshold Value for per trace length of IPC improvement

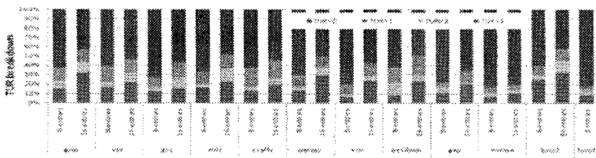

Figure 9 Trace utilization rate breakdown for a 8-entries trace cache and 16-entries trace cache

be access, and the threshold value is 16. The total trace size is 4KB, which achieve a high performance with small cache size.

7.3 RCM Performance

Figure 10 shows the IPC obtained by RCM for per benchmarks as well as the harmonic mean. Four different experiments performed are:

1) Using the retirement map table (RMAP) [18], the traditional state-reconstructing method.

2) Dual Path Instructions Processing (DPIP) [20]. Based on the table 1, we model the DPIP. At the same time, we have made the corresponding expansion for each stage of SimpleScalar pipeline in order to implement double program paths simultaneously.

3) RCM our proposed recovery mechanism.

4) IN_TRACE (INFINITE TRACE), in which a trace is made for every branch instruction, and assumes infinite resource.

As can be seen from Figure 10, RCM outperforms the other recovery mechanism arcs all benchmarks. RCM perform nearly as well as IN_TRACE.

The performance improvement compared with traditional processor comes from reducing the frond-end re-fill and flush latency. Comparing with the DPIP, The major contribution is that RCM prevent the instructions from non-critical path from entering the pipeline. Instead of forking the branch that is low-confidence, RCM only allows the branch that is low-confidence and enough critical can be forked a second path. Especially, miss-hit for fetching an alternative path (that actually is not critical) from the instruction cache, the penalty is horrible. In additional, the decoded instructions from the STC also implicitly reduce the burden of fetch stage that is just big bottleneck for modern processor. In all, average IPC improvement achieves 10.7% on RCM compared with a conventional processor without RCM, and 4.6% improvement for DPIP.

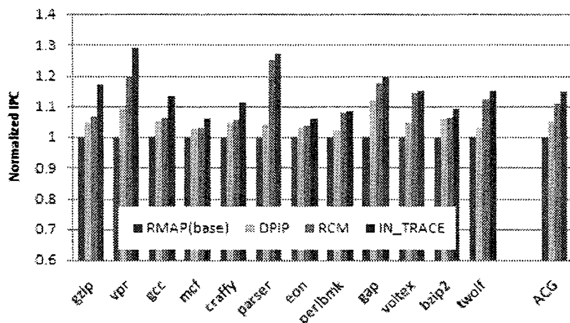

Figure 10 Comparison of four models performance

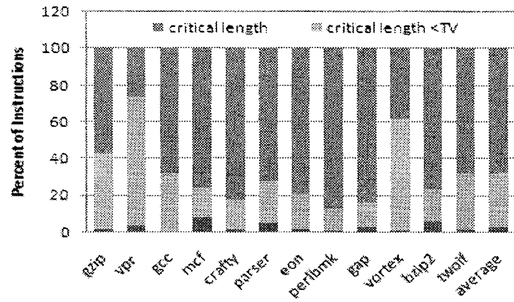

Figure 11 Breakdown of instruction critical for low-confidence prediction

7.4 Performance analysis

RCM over dual path mechanism can achieve high performance by preventing forked low-confidence prediction without enough criticality. Here, we also model the same critical predictor for DPIP to evaluate proportion of low-confidence prediction without enough critical in Figure 11. The figure shows the breakdown of instruction critical length for low-confidence prediction. Forking an alternative path on non-critical path or the critical path with its length less than TV can degrade performance even if the low-confidence prediction achieves success.

7.5 Instruction Window Size

This section studies the performance variation of the three approaches (the IN_TRACE model is removed) when the instruction window size and the reorder buffer size increase. Figure 12 shows the harmonic mean IPC when the instruction window size varies from 32 to 256. To focus the performance study on the RCM exclusively, the physical register file size is kept idealized in this group of experiments. As shown in Figure 12, all three models obtain performance improvement due to an increased instruction window size. However, the strides of the improvement are not equal. As can be seen, the performance gap between DPIP and RCM becomes larger as instruction window size increases. It is because that the ROB size is also the resource arbitration of the critical path predictor that is more exact following the ROB size increasing.

7.6 Effects of Pipeline Depth

Finally, we evaluated the effect of pipeline depth on the

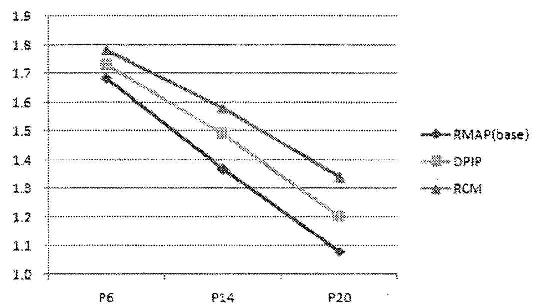

Figure 13 Average IPC for different pipeline depth

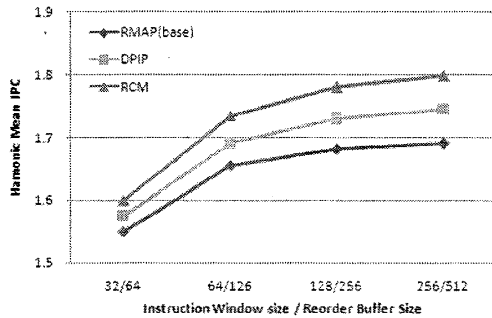

Figure 12 Performance of 3 models with different instruction / ROB size

performance of the each model. Figure 13 shows the average IPC for total pipeline of 6, 14, 20 stages. We can also see the strides of the improvement are not equal. For 6 stages, RCM obtains an average improvement of 9.8% over RMAP, but for 20 stages the obtained improvement are 26%. It is because the pipeline re-fill and flush are hidden. Further, the RCM uses the decoded instructions to increase the ILP of fetch stage and decode stage.

8. Conclusions

We proposed a recovery critical misprediction mechanism. It can reduce the latency of branch misprediction by hiding the re-fill penalty, reducing the burden of fetch process, and preventing the non-critical alternative instruction from entering pipeline by using critical path prediction. Different from previous double path methods, RCM need not to double the port to fetch alternative path (instead by rename stage that it is not performance bottleneck for data path), so reduce the complexity of hardware to achieve a higher frequency. The STC is a small simplicity cache (total size is 2KB/4KB). For some practical processor (32 KB L1 I-cache for MIPS R10000 [4] and 64KB L1 I-cache for Alpha 21264 [5]), the size of STC is satisfied. STC with small size increases the performance of processors, and the architecture of STC is greatly simplified comparing with common trace cache. According to our simulation results, RCM achieves a performance improvement by reducing the misprediction penalty. Using RCM, average IPC improvement achieves 10.7% up over the traditional recovery mechanism.

Acknowledgments

This research was supported by CREST, JST and partially by a grant of Knowledge Clusters Initiative 2nd stage, MEXT.

References

[1] E. Sprangle and D. Carmean. "Increasing processor performance by implementing deeper pipelines," In proceedings of the 29th Annual International Symposium on Computer Architecture, Pages 25-34, May 2002.
[2] C-C. Lee, I-C. Chen and T. Mudge, "The bi-mode branch predictor," MICRO-30, pp. 4-13, December 1997.
[3]T. Yeh, "Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors," year of Publication, 1993.G. Hinton, D. Sager, M. Upton, D. Carmean, A. Kyker, and P. Roussel. "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, February 2001.
[4]K. Yeager. "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, Vol 16, No.2, Page 28-40, April 1996.
[5]D. Leibholz and R. Razdan. "The Alpha 21264: A 500MHz out-of-order microprocessors," In Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON), pages 28-36, February 1997.
[6]K. Krewell. "IBM's Power4 Unveiling Continues", Microprocessor Report, November 2000.
[7]Heil. T. H, and Smith J.E, "Selective Dual Path Execution," University of Wisconsin-Madison, USA: Technical Report, ECE-98-8, 1996.
[8]P.N. Glaskowsky. "Pentium 4 (Partially) Previewed", Microprocessor Report, August 2000.
[9]Brian Fields, Shai Rubin and Rastislav Bodik. "Focusing Processor Policies via Critical-Path Prediction." The 28th International Symposium on Computer Architecture, 2001.
[10] E. Rotenberg, S. Bennett and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," In Proceedings of the 29th International Symposium on Microarchitecture, pp. 24-34, December 1996
[11] B. Black, B. Rychlik, and J. Paul Shen, "The Block-based Trace Cache," Proceedings of the 26th Annual International Symposium on Computer Architecture, page 196-207, May 1999
[12] E. Jacobsen, E.Rotenberg, J.E. Smith, "Assigning Confidence to Conditional Branch Predictions," Proc. 29th Annual Symp. And Workshop on Microprogramming and Microarchitecture (MICRO-29), 1996
[13] A. Klauser, A. Paithankar and D. Grunwald. "Selective Eager Execution on the PolyPath Architecture". Proc. of the Int. Symp. on Computer Architecture, 1998.
[14] D.Burgeer and T. Austin, "The simplescalar tool set. SIGARCH Computer Architecture," News 25, pp. 13–25, June 1997.
[15] AA. González, J.M. Llabería and J. Cortadella. "A Mechanism for Reducing the Cost of Branches in RISC Architectures". Microprocessing and Microprogramming, vol. 24,1-5, pp. 565-572, Aug. 1988.
[16] B. Calder. Simpoint. http://www-cse.ucsd.edu/~calder/simpoint/, 2006.
[17] Erez Perelman, Greg Hamerly and Brad Calder. "Picking Statistically Valid and Early Simulation Points," In Proc. Of the 2003 Intl. Conf. on Parallel Architectures and Compilation Techniques, pp. 244-255, September 2004.
[18] P. Zhou, S. Onder and S. Carr, "Fast Branch Misprediction Recovery in Out-of-order Superscalar Processors," ICS-19, pp. 41-50, June 2005.
[19] Scott McFarling, " Combining Branch Predictors, " TN 36, Compaq Computer Corporation Western Research Laboratory, June 1993.
[20] Juan L.A, Jose Gonzalez, A. Gonzalez, and James E. Smith, "Dual Path Instruction Processing", Proceedings of the 16th international conference on Supercomputing, pages: 220-229, 2002
[21] P.Y. Chang, M. Evers and Y.N. Patt. "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference". Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, 1996.
[22] L. Gwennap. "MIPS R10000 Uses Decoupled Architecture". Microprocessor Report, pp.18-22, Oct. 1994.
[23] W. W. Hwu and Y. N. Patt. "Checkpoint Repaire for out-of-order execution machines," In proceedings of the 14th Annual International Symposium on Computer Architecture, Pages 18-26, 1987.
[24] H. Akkary, R. Rajwar, and S. Srinivasan, "An Analysis of Resource Efficient Checkpoint Architecture," ACM Transactions on Architecture and Code Optimization (TACO), Volume 1, Issue 4, Dec. 2004.
[25] P. Zhou, S. Onder and S. Carr, "Fast Branch Misprediction Recovery in Out-of-order Superscalar Processors," *ICS-19*, pp. 41-50, June 2005.
[26] Eyerman, S, Smith, J.E. and Eeckhout, L, "Characterizing the branch misprediction penalty", Performance Analysis of Systems and Software, 2006 IEEE International Symposium , Page 48-58, Mar 2006.
[27] W.D. Connors, J. Florkowski and S.K. Patton. "The IBM 3033: An Inside Look". Datamation, pages 198-218, May 1979.
[28] J. Bondi, A. Nanda and S. Dutta. "Integrating a Misprediction Recovery Cache into a Superscalar Pipeline". Proc. of the Int. Symp. on Microarchitecture, 1996.