A two-phase recovery mechanism*

Zhaoxiang Jin
Michigan Technological University
Houghton, Michigan
zjin3@mtu.edu

Soner Önder
Michigan Technological University
Houghton, Michigan
soner@mtu.edu

ABSTRACT

Superscalar processors take advantage of speculative execution to improve performance. When the speculation turns out to be incorrect, a recovery procedure is initiated. The back-end of the processor cannot be flushed due to having a mixture of both valid and invalid instructions. A basic solution is to wait for all valid instructions to retire and then purge the invalid instructions. However, if a long latency operation, such as a Last-level Cache (LLC) miss appears before the misspeculation point, the back-end recovery time significantly increases.

Many proposed mechanisms selectively flush invalid instructions in order to speed up the back-end recovery. In general, these mechanisms rely on broadcasting some misprediction related tags to remove the instructions from any back-end structures, such as ROB, LSQ, RS, etc. The hardware overhead in these mechanisms is nontrivial and can potentially affect the processor clock cycle time if they are on the critical path. Moreover, a checkpointing mechanism or a walker needs to be added to accelerate the recovery of the front-end register alias table (F-RAT).

We propose a two-phase recovery mechanism which does not need any walking or broadcasting process and can still match the performance of the state-of-the-art recovery approaches. The first phase works similar to a typical basic recovery mechanism and the second phase is not triggered until the back-end is stalled by an LLC miss load. In that case, the second phase treats the load as a misspeculation and recovers from this load. Since the LLC miss response time is usually much longer than the time to fill the entire pipeline with new instructions, in most cases our mechanism can completely overlap the branch misprediction recovery penalty with the cache miss penalty.

CCS CONCEPTS

• **Computer systems organization** → *Superscalar architectures*;

ACM Reference Format:
Zhaoxiang Jin and Soner Önder. 2018. A two-phase recovery mechanism. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205300

*This work is supported in part by NSF grant CCF-1533828.

ICS '18, June 12–15, 2018, Beijing, China

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205300

1 INTRODUCTION

Speculative execution is a key component for modern processors to reach high performance. Predicting the branch direction and the target address [15] helps the processor to keep fetching subsequent instructions when the branch is unresolved. Value prediction [17] is used to overcome the data dependence limitation. Memory dependence prediction [20] is designed to bridge the in-flight stores and loads. With the given progress of speculative execution, we believe more and other types of speculation will be involved in the design of future architecture.

When the speculation is verified to be incorrect, a recovery procedure is triggered: I) The branch predictor's state is restored back to the misprediction point [12] so that the correct path instructions can be predicted properly; II) The fetch engine redirects to the correct path following the misprediction; III) The RAT (Register Alias Table) is reverted back to the misprediction point so as to correctly rename the new valid instructions; IV) The resources which are occupied by the incorrectly fetched instructions are released. These are the general tasks that any recovery mechanism should handle.

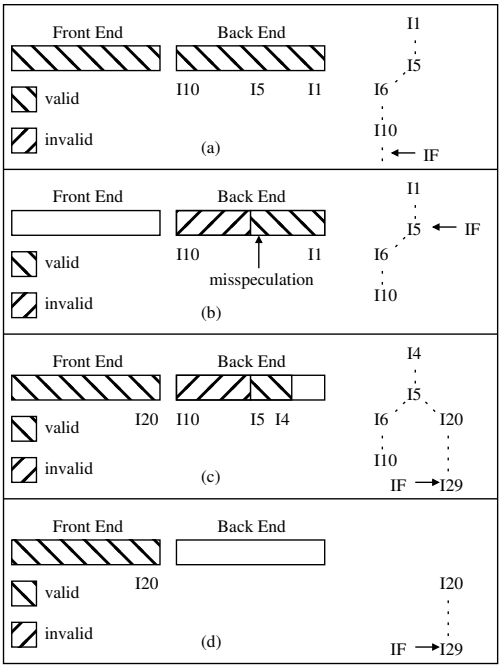


Figure 1: A basic recovery mechanism

A basic recovery mechanism, which is probably explained in every computer architecture book, is shown in Figure 1. The left hand side of the figure represents the instruction state in the pipeline. The

front-end includes fetch, decode, rename and dispatch stages. The back-end includes the Reorder Buffer (ROB), Reservation Station (RS) and Load/Store Queue (LSQ). The right hand side of the figure depicts the control flow graph (CFG) of the executing program fragment, including all the instructions currently in the pipeline.

Figure 1(a) shows the initial state at which *I1* is the oldest instruction in the back-end. *I5* is a branch which was predicted to take the left path followed by *I6*. This branch is found to be a misprediction in Figure 1(b). Therefore, the instructions following *I5* all become invalid and the fetch engine is rolled back to the mispredicted branch *I5*. Since the instructions left in the front-end are invalid, they can be cleared immediately. However, the back-end contains a mixture of valid and invalid instructions. Therefore, it can not be flushed. In Figure 1(c), *I1, I2* and *I3* are retired and new instructions starting with *I20* fill the front-end. Since the back-end has not fully recovered, the new instructions are not allowed to be dispatched to the back-end. Finally, in Figure 1(d), the last two valid instructions left in the back-end, *I4* and *I5*, are retired followed by a flush to clear the entire back-end. From this point on, the recovery procedure is completed and the new instructions will move to the back-end and start executing.

The time to recover the back-end is not necessarily longer than the time to fill the front-end with new instructions. However, if the back-end recovery is stalled by some long latency operation such as an LLC load miss, the penalty can be detrimental to the overall performance. As a result, several techniques were designed to address this issue specifically. We assessed these techniques and found that all of them require special storage and a broadcasting network to eliminate stale instructions dynamically. For example, in [11, 18], each unresolved branch has its own branch tag. The instructions have a set of branch tags indicating which branch they are dependent on. When a branch is mispredicted, the corresponding branch tag is broadcast to the entire pipeline which either selectively eliminates or marks its dependent instructions. Golander et al. uses checkpoint tags to selectively flush misprediction dependent instructions [8]. All of these mechanisms have to allocate renaming tags up-front for each instruction which may lead to a misspeculation, such as branches. On the other hand, if the instruction is not assigned a special tag, such as a load instruction, the corresponding misspeculation has to be recovered with a basic mechanism. Furthermore,the hardware cost of selective flushing is non-negligible. For every entry within the components in the back-end, a tag storage and a comparator needs to be added. A typical superscalar processor might have hundreds of these entries (Intel Skylake, 224 entry ROB, 97 entry RS, 72 entry Load Buffer, 56 entry Store Buffer). Furthermore, these comparator matrices consume additional energy during the recovery.

We argue that selective flushing speeds-up recovery only when the back-end recovery is blocked by some long latency operation and this is not the common case in real execution. Back in Figure 1, if the mispredicted instruction *I5* can be retired fast enough, the back-end can be reset without blocking the front-end. The reset process efficiently eliminates every instruction left in the back-end. In summary, when the back-end recovery time is short, the basic recovery mechanism works well. When the back-end recovery is long, a better recovery mechanism which is also cost-effective is desirable. We propose a two-phase recovery mechanism to optimize the recovery time for these two scenarios.

In our proposed mechanism, when a misprediction is detected, the first phase of the recovery is initiated. This phase is nothing but the basic recovery mechanism shown in Figure 1. The second phase will not be triggered until the back-end recovery procedure is stalled by a long latency operation. Among all the long latency operations, loads which miss in the LLC are the most detrimental to the back-end recovery. Therefore, when the ROB head reaches an LLC miss load, the second phase of recovery is triggered. The second phase simply treats the missing load as if it is a load misspeculation and restarts fetching the stream from the load. When the newly fetched stream arrives at the end of the front-end, the back-end is reset and the recovery procedure terminates. During the second phase of recovery, the load and the following instructions left in the back-end cannot be retired nor update any architectural state. When the mispredicted instruction which triggered the recovery procedure is fetched again during the second phase of recovery, instead of the predicted values, the correctly computed values are substituted. For example, a mispredicted branch will use its computed result instead of the prediction provided by the branch predictor.

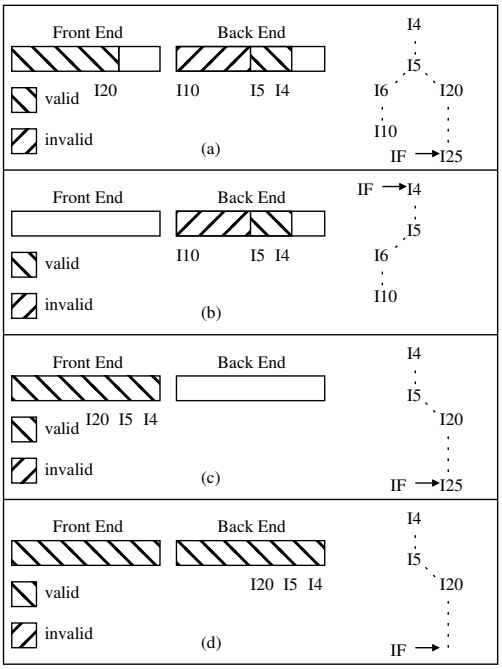


Figure 2: The second phase of the recovery

Figure 2 illustrates the second recovery phase. The first phase is shown in Figure 2(a) in which *I4* is an LLC miss load which stalls the retire stage. This load triggers the second phase which is shown in Figure 2(b). The front-end is flushed again and this time the fetch engine is redirected to the instruction which stalls the back-end, *I4*. When the mispredicted instruction *I5* is fetched, it uses its computed result and takes the right path. Figure 2(c) shows that when the front-end is filled with the correct instructions, the back-end is reset and the recovery procedure terminates. If the

memory response time of *I4* is long enough, the whole pipeline is stalled in Figure 2(d). When that happens, the branch misprediction recovery penalty completely overlaps with the cache miss delay.

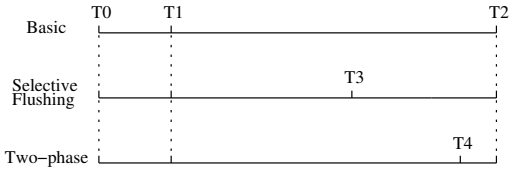


Figure 3: Timeline of Recovery

For the three recovery techniques, basic, selective flushing and two-phase, the recovery timeline is shown in Figure 3. **T0** is the point at which the misspeculation is detected and the recovery procedure commences. **T1** is the time when ROB head reaches the cache miss load and **T2** is the time the cache miss load obtains its data from the memory. For the basic mechanism, from **T0** to **T2**, the processor cannot dispatch new instructions to the back-end as the back-end has not recovered. In case of selective flushing mechanism, we assume the flushing operation takes place instantly. Therefore, it takes **T0**∼**T3** to fetch and dispatch the new stream starting from the misprediction point and fill the back-end. For the two-phase mechanism, the second phase does not launch until **T1** is reached. Thereafter the processor takes **T1**∼**T4** to fetch and dispatch instructions starting from the cache miss load and filling the back-end. Since the two-phase mechanism dispatches more instructions, (**T4-T1**) is greater than (**T3-T0**). As long as the cache miss latency is longer than this time, the performance of selective flushing and the two-phase recovery would be identical.

2 PRELIMINARY ANALYSIS

We have explained the reasoning behind why this two-phase recovery mechanism should work well. In this section, we present a preliminary analysis to quantitatively assess the differences between the proposed mechanism and other techniques.

Table 1 lists the recovery related statistics. The evaluation is accomplished by simulating a basic recovery mechanism and the configuration is specified in Section 5. The first part of the data shows the number of mispredictions per 1k retired instructions (MPKI). The data is divided into two subcategories, the back-end recovery stalled by missing loads and the recovery which is not stalled. Clearly, those cases which are not stalled represent the common case and we conclude that the basic recovery mechanism should work well in most cases. The second part of the data shows the number of cycles spent between the misprediction detection and retirement of the mispredicted instruction, namely, the back-end recovery time. This result is also divided into two subcategories. Note that the back-end recovery has a very large delay when it is stalled by cache miss loads except in *lib* and *h264ref*. The average delay is 105.3 cycles through all simulated benchmarks. On the other hand, the average recovery delay is only 4 cycles for those which are not blocked by a cache miss. Given that modern superscalar processors have 10 pipeline stages or more to fetch, decode, rename and dispatch instructions, the back-end recovery delay will be mostly overlapped by the front-end fill delay, if the back-end is not stalled by a cache miss.

Table 1: The average misspeculation recovery ratio and the corresponding back-end recovery time

	misprediction ratio		average cycles	
	w/ miss (MPKI)	w/o miss (MPKI)	w/ miss (Cycles)	w/o miss (Cycles)
perl	0.1972	7.6372	140.7410	4.1971
bzip2	0.2008	10.7664	87.5747	3.3479
gcc	0.1965	2.0736	121.0998	4.0351
mcf	2.2920	7.0885	117.3951	2.9718
gobmk	0.0900	11.4311	90.9878	4.0375
hmmer	1.1071	10.0676	101.7260	2.8986
sjeng	0.0668	8.9227	121.4520	4.2428
lib	0.0445	0.0234	7.9040	2.0131
h264ref	0.1252	1.6037	27.2903	4.9313
astar	0.5413	10.7609	84.3921	2.5816
bwaves	0.0254	0.4362	113.1180	2.2977
milc	0.0002	0.3065	145.5395	2.3869
zeusmp	0.0329	2.4058	112.1057	2.8289
gromacs	0.0011	28.6441	104.3921	2.2456
leslie3d	0.0164	12.0679	125.6261	2.1740
namd	0.0066	1.2174	93.2071	4.9071
Gems	0.0024	0.0064	124.3514	20.4624
tonto	0.0099	4.0393	136.5524	2.9060
lbm	0.0061	0.0103	141.6629	2.0402
wrf	0.0116	1.0228	125.1721	3.1938
sphinx3	0.3948	1.3044	89.9858	3.3013
Average	0.2557	5.8017	105.3465	4.0000

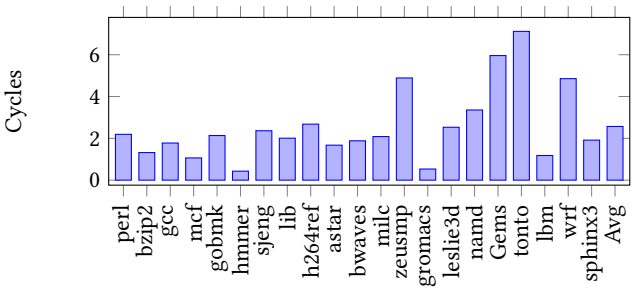


Figure 4: Phase one to Phase two delay

Figure 4 shows the average number of cycles spent between misspeculation detection and the point when the back-end is blocked by a long latency instruction. In our simulation, the long latency instruction is an LLC miss load. The corresponding delay is equivalent to the data shown in Figure 3 as **T0**∼**T1**. It is clear that if the recovery procedure is blocked by a cache miss, the time to arrive at that point is very short. The average delay through all benchmarks is only 2.57 cycles. Moreover, *hmmer* and *gromacs* have smaller than one cycle delay, which means in most cases the back-end has already stalled when the misprediction is detected.

Figure 5 illustrates the number of instructions left in the ROB between the misprediction and the missing load. This number varies from 8.43 to 201.35 and the average is 86.92. The basic recovery

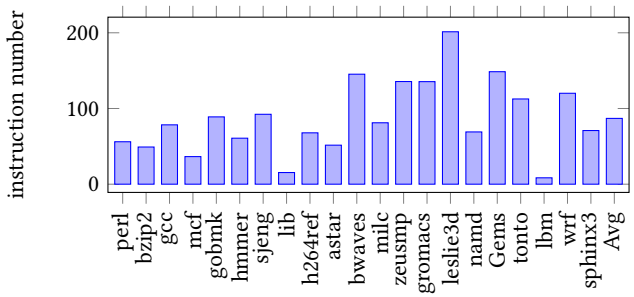


Figure 5: The number of instructions between the cache miss load and the misprediction

mechanism we simulated is an 8-issue superscalar processor. Approximately, 10.86 cycles are used to fetch the extra instructions compared to a selective flushing mechanism. When we combined this delay with the time used to enter phase two recovery in Figure 4, the average number of additional cycles is 13.4. This is the extra delay for our two-phase recovery mechanism, compared against an architecture which can instantly eliminate invalid instructions. This delay can also be found in Figure 3 as **T4** - **T3**.

3 RELATED WORK

3.1 Selective Flushing

Before we discuss the details of the microarchitecture implementation of our two-phase recovery mechanism, we visit the related work in this area. Broadly speaking, we can classify the selective invalidation mechanisms into two categories, invalidation before the mispredicted instruction is retired and invalidation after the mispredicted instruction is retired. The former aims to eliminate the stale instructions before dispatching new instructions. The latter can dispatch new instructions even when the back-end is mixed with valid and invalid instructions. Therefore, the front-end is not restricted by the back-end recovery time. When the mispredicted instruction is retired, selective flushing can eliminate the stale instructions.

Detailed data on real implementations of recovery mechanisms in existing processors is scarce as most corporations do not disclose their recovery techniques about reducing the front-end stall time. However, from published patents we have an approximate idea about their designs. Kyker et al. associated each instruction with a path color [14]. When a misprediction is detected, the new instruction stream is assigned a different path color. Hence, the new instruction stream can be dispatched to the back-end even if it has not cleared. At the retire stage, the instructions which belong to the old path are retired until the mispredicted instruction is reached. Thereafter, the back-end contains stale instructions belonging to the old path and the valid instructions belonging to the new path. The stale instructions are retired without updating any architectural state. In order to accelerate the process of retiring stale instructions, several approaches are discussed. One of these mechanisms is to broadcast the old color to eliminate every instruction on the old path simultaneously. The second is to give the old path instructions priority to execute regardless of their operand values.

We argue that any mechanism which eliminates stale instructions after the mispredicted instruction is retired can delay the elimination process for ROB and RS, but not for LSQ since it is searched associatively. For example in Figure 6, a new path (blue) is assigned when the misprediction in the old path (red) is detected. The instructions between the head and the misprediction are valid and the rest of the instructions (red) are stale. If the new load instructions (blue) are dispatched to the damaged LSQ, they may receive incorrect values. In this example, correct execution requires forwarding of the value from **SW1** to **LW**. However, **SW2** is closer and its value may be incorrectly forwarded to **LW**. Since **SW1** and **SW2** have the same path color, there is no way for the processor to distinguish them. In summary, the LSQ needs to be repaired or at least marked differently before any new memory instruction are dispatched.

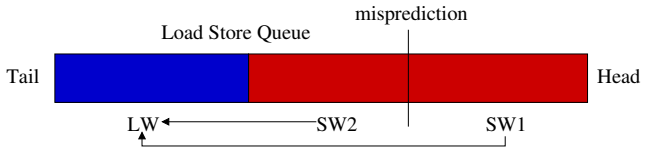


Figure 6: The issue to dispatch new instructions into the LSQ when it is not fixed

McIlvaine et al. proposed a tag broadcasting mechanism to selectively eliminate stale instructions in the back-end [18]. This mechanism is also very similar to Mower [11]. Each branch is assigned a branch flush tag and the tag positionally encodes branches by setting only a single bit. Therefore, the number of tag bits constrains the number of unresolved branches. In addition to the branch flush tag, a branch path tag is created and assigned to each renamed instruction. The branch path tag is the logical aggregation of unresolved branch flush tags. When a branch misprediction is detected, the instruction which has the corresponding bit set in its branch path tag will be flushed. The branch flush tag bit is reclaimed when the particular branch is resolved. A similar mechanism was also proposed for a checkpoint processor [8] except that the branch flush tag is substituted with the checkpoint tag. Another difference is the instructions fetched after the misprediction may belong to multiple checkpoints so the flushing operation is accomplished by broadcasting these checkpoint tags sequentially.

These approaches can identify the stale instructions by assigning different tags upfront, which is better than coloring the path. Nevertheless, the tag has to be allocated before the speculation is initiated. If a mispredicted instruction does not have a tag, it will further complicate the design. Our mechanism does not need any special tags and can recover from any kind of misspeculation.

3.2 RAT recovery

The Register Alias Table (RAT) is used to rename instructions to eliminate false data dependencies. During recovery, any updates made to RAT after the misprediction have to be undone. The basic recovery mechanism incorporates an additional RAT. We refer to the map table used for renaming as the front-end RAT (F-RAT) and to the one used at the retire stage as the Retirement RAT (R-RAT).

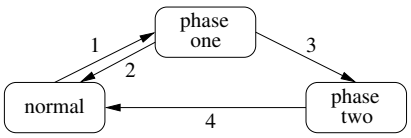
R-RAT is updated by retired instructions. During the recovery, when a mispredicted instruction is retired, R-RAT contains the correct register mapping up to that point. Subsequently, R-RAT is copied onto F-RAT and the new instructions are renamed using this corrected F-RAT afterwards [9].

Other than this basic recovery mechanism, the MIPS R10000 and Alpha 21264 processors used checkpoints to restore F-RAT [13, 26]. Zhou et al. proposed EMR (Eager Misprediction Recovery) to take the checkpoint of F-RAT when the misprediction was detected [27]. When the correct F-RAT was recovered, the differences between the checkpoint and F-RAT were the misprediction affected mappings. Thereafter, EMR would insert **MOV** instructions to forward the correct data to the affected physical registers. Other mechanisms, such as walking, can incrementally repair F-RAT by walking through the ROB [1]. The walking process can walk from either the ROB head by using R-RAT or the ROB tail by using F-RAT. F-RAT is not recovered until the walking process reaches the misprediction. Another special walking mechanism was proposed in Mower [11]. Mower has a bit vector to indicate which logical register is defined after the misprediction and mark them as invalid. Therefore, the renaming procedure continues if an instruction only consumes valid register mappings. In the recovery, Mower walks from the misprediction to the ROB tail and incrementally repairs F-RAT. Mower may potentially recover F-RAT faster than other walking mechanisms.

ROB-free mechanisms were primarily designed to scale up the instruction window [1, 2]. One of these mechanisms is the checkpointing architecture in which each checkpoint is in charge of a group of instructions. Once every instruction in the checkpoint is executed, the checkpoint can be retired. If the mispredicted instruction has a checkpoint, F-RAT can be restored. Otherwise, the recovery process had to roll back to a prior checkpoint. Cristal et al. further optimized this checkpointing architecture [5, 6]. In this technique, a pseudo-ROB was implemented to reduce the number of necessary checkpoints. This pseudo-ROB could retire instructions disregarding their completion status and only assigned a checkpoint if a retired branch was not executed.

4 MICROARCHITECTURE

In this section, we elaborate on the microarchitecture details of our two-phase recovery mechanism. Figure 7 demonstrates the state machine diagram of the mechanism. The phase one of the recovery is so pervasive that we will not describe it. We are going to focus on the explanation of the phase two recovery.



1. A misprediction is detected.
2. The misprediction is retired, the back–end is flushed.
3. The ROB head is stalled by a cache miss load.
4. The mispredicted instruction is re–fetched and the invalid instructions are flushed.

Figure 7: Two phase recovery state machine

Initialization. The first step of the second phase is to detect a cache miss load. We adopted the design from Intel's P6 microarchitecture in which load instructions are kept in a load buffer. Every issued load will receive a feedback at the last cycle of its access, either the required data or a cache miss message. The load buffer maintains these information and when a cache miss load blocks the ROB head, the phase two recovery is initiated.

First, the fetch engine is redirected to the PC of the load which blocks the ROB head. The instructions left in the front-end and the back-end are all treated as invalid instructions. Hence, they are not allowed to retire nor update any architectural state. The state of the branch predictor also needs to be recovered. The most important component is the branch history register (BHR) which contains the previous branch predictions. Usually, the BHR has more history bits than what a branch predictor requires [12]. As a result, the processor can shift back to the misprediction point during the recovery. Each branch instruction has a pointer pointing at the BHR position when the branch is predicted. So if the branch is mispredicted, its pointer is used to recover BHR. In order to recover BHR during phase two recovery, we need a new pointer to keep the location of the most recent retired branch. It is the retirement BHR pointer (R-BP) which works similar to R-RAT. Whenever a branch is retired, its BHR pointer is copied to R-BP. At the beginning of the phase two recovery, R-BP is used to recover BHR. Another very important component is the return stack buffer (RSB) which is used to provide the return address after a function is called. It is possible that RSB is corrupted if the branch inside the function is mispredicted and the return address in RSB has been overwritten. A renamed version of RSB [7] is implemented to overcome this problem, hence the processor can restore RSB after any misprediction. We implement a retirement mapping table for RSB similar to R-RAT. Therefore, RSB can be repaired in phase two of the recovery.

Since instructions left in the pipeline are invalid, they can be immediately evicted during phase two recovery. However, in many cases the two paths emanating from a mispredicted branch instruction converge. As a result, even after a misspeculation was detected, the execution of invalid instructions might still warm up the data cache correctly. Consequently, we have three different flushing policies. The *conservative policy* is to flush both the front-end and the back-end, which wastes the least energy to execute stale instructions. The *moderate policy* only flushes the front-end so that stale instructions left in the back-end are allowed to execute. The *aggressive policy* does not remove any instructions and keeps dispatching stale instructions left in the front-end until the back-end fills up. If the dispatch stage is stalled by stale instructions, these instructions left in the front-end are removed. In this policy, the front-end is mixed with newly fetched valid instructions and stale ones. A one bit coloring mechanism is thus added. Once valid instructions reach the back-end, all stale instructions are eliminated irrespective of the policy that is used.

Fetch Policy. In phase two recovery, the misspeculated instruction will be re-fetched and re-executed. If the default prediction is used, this instruction will be mispredicted again and cause a deadlock. Therefore, the execution result should be used to overwrite the prediction for mispredicted instructions.

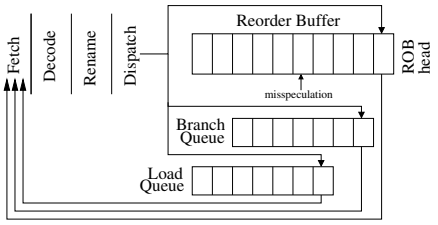


Figure 8: Overwriting the prediction

Figure 8 illustrates the overwriting mechanism. The task of finding the first misspeculation is done by scanning the ROB entries. Each ROB entry has one bit indicating the misspeculation status. Every fetched instruction has to read its corresponding entry in the ROB and if the entry has the misspeculation bit set, the executed result is reused instead of the prediction. From this point on, no instructions are reused. The scan operations are already implemented in the basic recovery mechanism, thus, there is no extra overhead. The misspeculation bit is set when the instruction is executed and reset when the ROB entry is assigned to a new dispatched instruction.

We keep the correct branch direction and the branch target for branches in the branch queue. If the first misspeculation is a branch, the fetch unit reuses the executed result from the branch queue instead of the prediction from the branch predictor. For a load, the colliding store is kept in the load queue. If the first misspeculation is a load, the colliding store is added and the dependent load will not execute until the store has executed. Since the same misspeculation information is maintained in the basic recovery mechanism, no additional overhead is incurred. Other types of misspeculations can be recovered in the same way.

F-RAT and free register pool. While the new instruction stream reaches the rename stage, F-RAT is repaired by copying from R-RAT. This is exactly the same procedure that is used in the basic recovery mechanism. There is no need to take checkpoints or implement a walking process. The free register pool is repaired in the same way as the basic recovery mechanism.

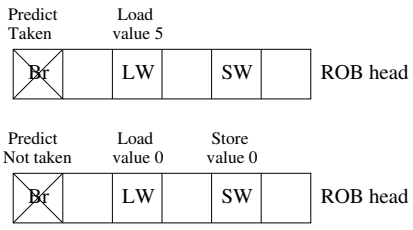


Figure 9: Misprediction due to speculative recoveries

Speculative recovery. Most recovery mechanisms recover in program order such that the oldest misspeculation is repaired. In our mechanism, we assume the instructions between the misspeculation and the ROB head do not cause any more misspeculations during phase two recovery. Therefore, it is speculative and has its own side effect. For example, the top half in Figure 9 shows a branch which was predicted taken. This branch is mispredicted as one of its operands comes from the load and the load reads a value of 5. Thus a recovery procedure is initiated and finally a phase two recovery is issued. The bottom half shows the instruction execution after recovery. The load has the same address as the store which was not executed before the recovery. This time the load is forwarded from the store through the store queue and gets a value of zero. As a result, the branch is mispredicted again. The fact is, although the original prediction was correct, due to the load misspeculation, this is not correctly detected. Therefore, the branch instruction ends up triggering the recovery procedure twice. Our evaluation shows that such cases are rare (fewer than 0.01% of the total misspeculations), so their impact is negligible.

Complexity Comparison. The hardware complexity should be taken into account when we are designing a recovery mechanism. So, we therefore compare the overhead among different techniques.

The first part is the overhead to eliminate stale instructions. The selective flushing mechanism which uses the branch path tag [18] has to keep a tag for every entry in the back-end, such as ROB, RS, LSQ, etc. Whenever a branch is resolved, a single bit is broadcast to reclaim the bit in the branch path tag. If the branch is mispredicted, another bit is broadcast to selectively flush the entries which have the corresponding tag bit set. Figure 10 shows a branch path tag with three bits which are implemented by D-type registers. Initially, the tag was 111 which means the corresponding instruction was dependent on branch 001, 010 and 100. The right side of the figure shows the waveform of the simulation. In the first cycle, branch 001 is resolved and it broadcasts X[0] to reclaim the bit. In the second cycle, branch 010 resolves and it is mispredicted. Therefore, the misprediction bit is set. Since the tag indicates this instruction is dependent on the branch 010, then the "reset entry" signal is triggered to flush the corresponding entry. The number of the D-type registers is determined by the maximum number of unresolved branches and the number of the entries in the back-end.

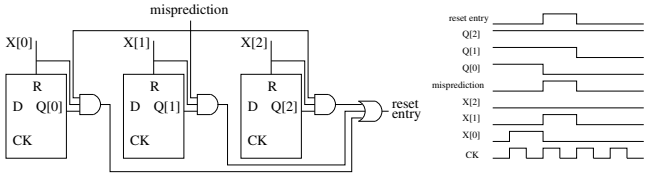


Figure 10: The branch path tag for selective flushing

The second part is the overhead to recover F-RAT. We assessed two commonly used mechanisms, checkpointing and walking. Each checkpoint has to have the same amount of storage as F-RAT. When a checkpoint is allocated, the copy process from F-RAT to the checkpoint has to be done in one cycle or extra delay will be incurred at the rename stage. Since keeping a checkpoint for every branch is costly, a selective checkpoint allocation algorithm is widely used [2]. In which case, a confidence predictor is required and its accuracy will have an immense impact on the utilization of checkpoints. On the other hand, walking process needs at least an additional RAT to walk with. At the beginning of the recovery, this RAT is copied from R-RAT or F-RAT depending on the walking direction. The

walker then walks through the ROB entries to collect the updates of register mappings. If the ROB head is retiring instructions and updating R-RAT at the same time, a second read port to the ROB is necessary for the walker.

Our two-phase recovery mechanism reuses most of the resources needed for a basic recovery technique. The only difference is a two-phase state machine and another detection structure to overwrite the prediction of mispredicted instruction. Table 1 shows that less than 5% of the recoveries were obstructed by LLC miss loads. The total overhead to optimize these 5% cases should be minimized.

5 SIMULATION METHODOLOGY

We simulated our mechanism by using MIPS-I ISA without delayed branching. This ISA is very similar to PISA ISA, used by SimpleScalar [3]. GCC 4.9.2 tailored to this ISA is used to compile the benchmarks and generate binary code with the highest optimization ("-O3") set. We choose Spec 2006 as our benchmark suite. All simulation models were designed with Architecture Description Language (ADL) [21]. The ADL compiler can automatically generate the assembler, the disassembler and a cycle-accurate simulator which respects timing at the register transfer level from the description of the microarchitecture and its ISA specified in ADL language.

In order to efficiently simulate our mechanisms, we incorporated Simpoint 3.2 [22, 25] to minimize the simulation time. For each benchmark, a set of checkpoint images were generated where each checkpoint image contains the complete memory data segments, the register file and the program counter (PC). Other architecture related structures were not included, such as cache, branch predictor, memory dependence predictor, etc. Hence, the simulation of each interval has a cold start. In order to compensate for this effect, we selected a large size, 100 million retired instructions, to simulate each interval. Since each interval simulation was independent of others, we simulated all of the intervals simultaneously to further shorten the simulation time. Currently, the file descriptors are not kept in the checkpoint. Therefore, we cannot simulate an interval if it has file operations and the file descriptor was created before the checkpoint. When this happened, we replaced that checkpoint interval with the dominant checkpoint in that benchmark. In *h264ref*, we substituted one checkpoint (0.92% weight) with the dominant checkpoint (18.14% weight) and in *hmmer*, we substituted two checkpoints (0.22%, 0.43% weight) with the dominant checkpoint (98.9% weight). As the replaced intervals have very limited weights (<1.0 %), we expect the impact of this substitution to be negligible. We also modified McPAT 1.4 [16] to evaluate the dynamic energy consumption. DRAMSim2 [23] was also embedded to evaluate the memory subsystem behavior.

The benchmarks we simulated in this work are:

Integer: *perl*, *bzip*, *gcc*, *mcf*, *gobmk*, *hmmer*, *sjeng*, *lib*, *h264ref*, *astar*.

Float: *bwaves*, *milc*, *zeusmp*, *gromacs*, *leslie3d*, *namd*, *Gems*, *tonto*, *lbm*, *wrf*, *sphinx3*.

We simulated these benchmarks with the "ref" input. The remaining missing benchmarks were not included due to our linker's inability to link them. The configuration shown in Table 2 was shared by all the simulation models. The implementation details of different recovery mechanisms are as follows:

Table 2: Processor Configuration

ROB / RS / PRF	256 / 64 / 320
Fetch / Decode / Issue	8 / 8 / 8
Cache	32KB 8-way set associative iL1; 32KB 8-way set associative dL1; 512KB 8-way set associative L2, 10 cycles hit latency;
Memory	16GB DDR3L-1600, 2 channels, 2 ranks, 8 banks, open page, up to 64 pending requests [19]
Recovery Penalty	minimum 15 cycles
Int ALU / Int Mul	1 cycle / 3 cycles
Int Div, FP ALU	7 cycles
Branch Predictor	8 kB TAGE [10]
Memory Ordering	Store Sets [4]
Tech node	22nm
Clock frequency	3.2GHz

1. Recover Afterwards: In this model, the stale instructions are handled after the misprediction is retired. Path information [14] is carried by every instruction. The new instruction stream fetched after a misprediction was detected is assigned a new path color. So after the mispredicted instruction retires, the remaining instructions belonging to the wrong path are invalid. A walker is implemented to recover F-RAT walking from the ROB head towards the misprediction. The stale instructions are retired without updating any architectural state. There are two methods to retire such instructions sooner. One is to handle them as they execute. We refer to this policy as *Recover Afterwards without executing Bogus instructions* (RA-WOB). The other is to execute a stale instruction unless it is a LLC miss load where the load is forwarded with a bogus value. This policy is called *Recover Afterwards with executing Bogus instructions* (RA-WB).

2. Recover Beforehand: This architecture selectively flushes the stale instructions before the misprediction is retired. A branch flush-tag is assigned to the branch when it is renamed and a checkpoint of F-RAT is taken with this branch. If this branch is mispredicted, its tag is used to evict the stale instructions and its checkpoint is used to recover F-RAT. There are two different configurations to assign flush-tags. One is to greedily assign whenever the flush-tags are available. The other is to assign tags only for the low confidence branches. We implemented a confidence predictor [24] within the internal storage of the branch predictor. The two policies we evaluate are *Recover Beforehand with Greedy Allocation* (RB-X-GA) and *Recover Beforehand with Low confidence Allocation* (RB-X-LA). The 'X' represents the number of flush-tags which is also equal to the number of checkpoints. Both configurations keep dispatching branches with no checkpoints when checkpoint space is not available. Note, these approaches cannot recover other types of misspeculations and are limited to branches which have flush tags. A basic recovery mechanism is used for these misspeculations.

3. Two-Phase: This is our mechanism which can recover from any type of misspeculation and does not need special tags or checkpoints.

4. Inf: This simulation model has an infinite number of checkpoints. Therefore, it can recover from any type of misspeculation.

6 SIMULATION RESULTS AND ANALYSIS

Recover Afterwards and Two-Phase. Figure 11 shows the IPC results normalized to the Inf configuration for RA-WOB, RA-WB and Two-Phase. Using geometric means, the results in RA-WOB, RA-WB and Two-Phase are 0.988, 0.998 and 0.994 respectively. In the figure, RA-WB outperforms the architecture with an infinite number of checkpoints in *gcc* and *mcf*. The reason is RA-WB can execute stale instructions and is able to warm up the cache. However, Inf has to flush stale instructions and the future loads experience a longer latency. From our experiments, the average load latency in RA-WB is 36.6 cycles for *gcc* and 104.7 cycles for *mcf*. The latency in Inf is 37.7 cycles for *gcc* and 108.1 cycles for *mcf*.

RA-WB surpasses RA-WOB in most benchmarks except in *bzip2* where executing stale instructions negatively affects performance. Stale instruction execution may warm up the cache, but it can also evict some useful cache lines. From our experiments, the average load latency for *bzip2* is 26.9 cycles in RA-WOB and 28.7 cycles in RA-WB. It is clear that more useful cache lines were evicted by stale instruction execution in *bzip2*.

Using geometric means, RA-WB works better than Two-Phase, but only marginally. We argue that most of the benefits are contributed by a single benchmark, *mcf*. Therefore, we calculated the Gmean again excluding *mcf* and Two-Phase was about 0.1% better than RA-WB. The performance of RA-WB is highly dependent on the behavior of stale instruction execution. We simulated an architecture with the basic recovery mechanism and RA-WB performed even worse than the basic recovery mechanism in *bzip2*. On the other hand, Two-Phase consistently worked better than the basic recovery mechanism in all benchmarks. Two-phase outperformed the basic recovery mechanism by up to 8.35% (astar). The Gmean is 3.05% for Int and 0.4% for FP.

Recover Beforehand and Two-Phase. Figure 12 shows the IPC for RB-8-GA, RB-8-LA and Two-Phase normalized to the Inf models. The Gmean is 0.995, 0.992 and 0.994 respectively. The performance gap between a checkpointing architecture and Inf is caused by misspeculations which are not covered by checkpoints. We collected the number of misspeculations which have checkpoints in RB-8-GA, shown in Table 3. The "total" column means the total number of Misspeculations Per 1k retired Instructions (MPKI), including branch misspeculations and memory ordering misspeculations in our simulation. The "ratio" column represents the percentage of misspeculations which have checkpoints. In RB-8-GA, only branches are assigned checkpoints. As shown, a very large number of misspeculations are assigned checkpoints in *hmmer*, thus the performance is very close to Inf. On the other hand, *perl*, *mcf* and *astar* had fewer misspeculations covered by the checkpoints so the performance gap enlarges.

The performance gap between Two-Phase and Inf in Figure 12 is due to the extra delay in phase one and phase two of the recovery. In phase one, even if the ROB head is not blocked by any LLC cache miss load, retiring instructions before the misprediction may still take a long time. If the time to retire these instructions is longer than the time it takes for the valid instructions to go through the front-end, the front-end is stalled. In phase two, it takes extra cycles to enter phase two (Figure 4) and additional cycles to re-fetch the valid instructions before the misprediction (Figure 5).

Table 3: The misspeculations which have checkpoints

	total (MPKI)	ratio		total (MPKI)	ratio
perl	7.9478	88.48%	bzip2	10.9672	89.85%
gcc	2.2935	85.64%	mcf	9.3805	77.87%
gobmk	11.5669	94.08%	hmmer	11.1747	96.86%
sjeng	9.0844	87.42%	lib	0.0679	42.00%
h264ref	1.7351	90.45%	astar	11.3023	93.09%
bwaves	0.4615	97.23%	milc	0.3066	99.07%
zeusmp	2.4386	97.35%	gromacs	28.6453	99.78%
leslie3d	12.0843	98.04%	namd	1.224	98.47%
Gems	0.0088	99.93%	tonto	4.0508	98.20%
lbm	0.0163	99.90%	wrf	1.0344	97.40%
sphinx3	1.6993	89.92%	Average	6.0710	91.48%

Figure 13 depicts the extra cycles incurred by Two-Phase for every 1k retired instructions. This delay is divided into two parts, phase one and phase two. Apparently, the delay in phase two dominates the total delay. In the figure, *mcf*, *hmmer* and *astar* have the most extra delays, thus Two-Phase has the largest performance gap in these benchmarks compared with Inf.

Allocation Algorithms in Checkpointing Architectures. Figure 14 illustrates the Gmean of architectures with different checkpoint sizes and different allocation algorithms, normalized to Inf. Note that with 4 checkpoints, the low confidence allocation algorithm works slightly better than the greedy algorithm. But in case of 8 checkpoints, the greedy algorithm works better. This is because when the resources are constrained, a more efficient allocation algorithm is better. Otherwise, allocating resources greedily is better as the low confidence allocation may not fully utilize the available resources.

Memory latency effect. The memory accessing latency has a substantial impact on the overall performance and the average latency is 137.5 cycles through all the benchmarks. We simulated our Two-Phase algorithm using different memory frequencies and the results are shown in Figure 15. Note that Two-Phase gets closer to Inf as the memory latency increases. From the timeline in Figure 3, we learned that it is more likely for Two-Phase to have the same state as Inf if the memory latency is longer. As a result, more recovery penalties overlap with cache misses.

ROB size effect. Figure 16 shows the performance of our Two-Phase algorithm using different ROB sizes, normalized to Inf. The performance gets closer to Inf when the ROB size is reduced. The reason is similar to the case of memory frequency. A smaller ROB is much easier to fill when a load misses in the cache and blocks retirement. Therefore, more branch mispredictions overlap with the cache miss latency. On the other hand, Inf can fetch and execute instructions earlier than Two-Phase when the pipeline is not stalled with a larger ROB.

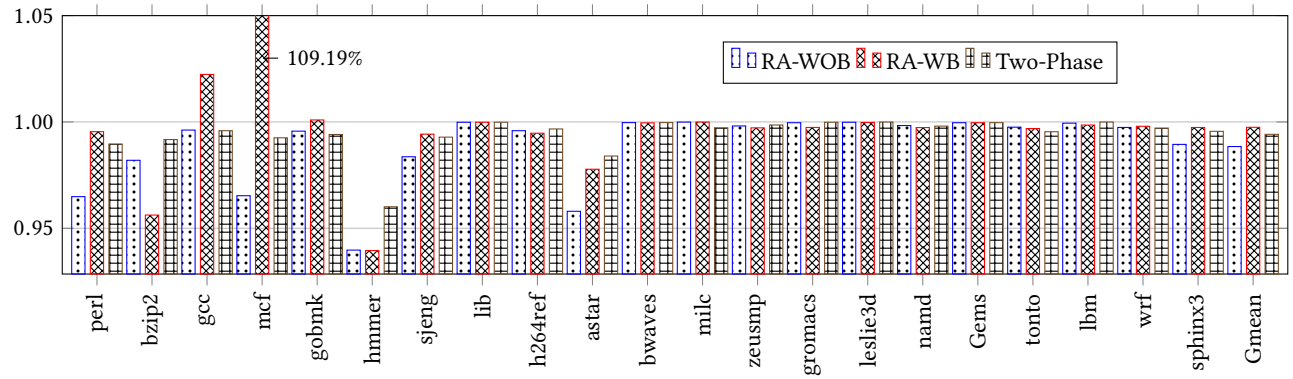


Figure 11: Recover Afterwards with/without executing bogus instructions vs. Two-Phase, normalized to Inf

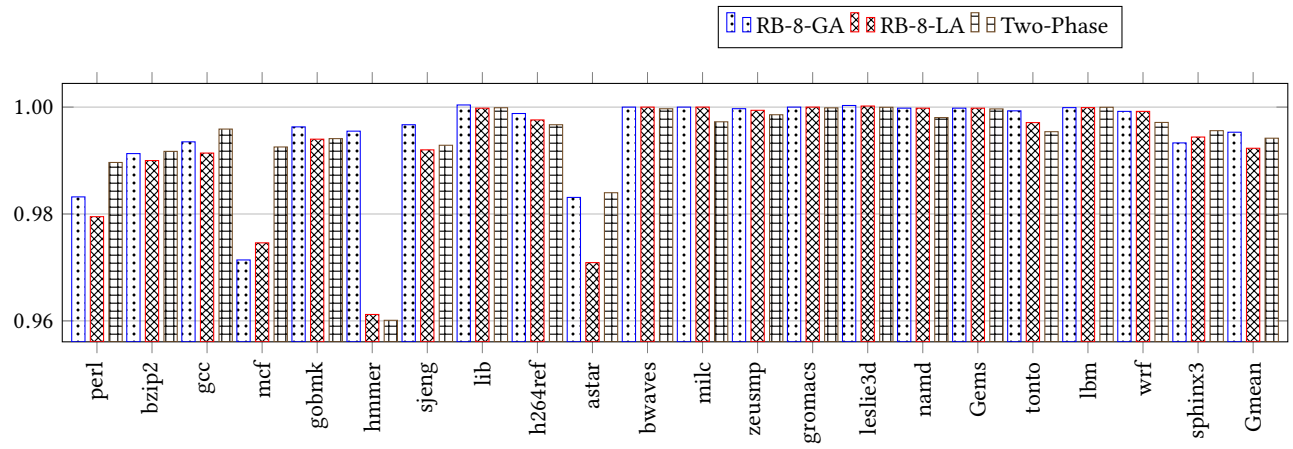


Figure 12: RB-8-GA vs. RB-8-LA vs. Two-Phase, normalized to Inf

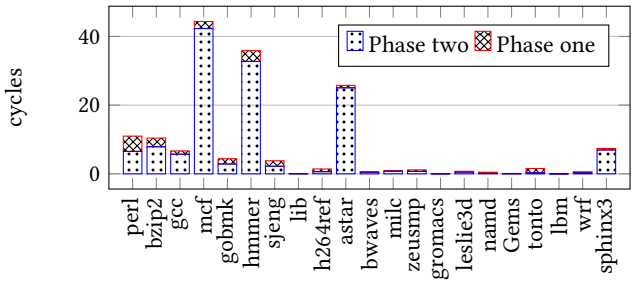


Figure 13: The extra cycles caused by Two-Phase compared with Inf

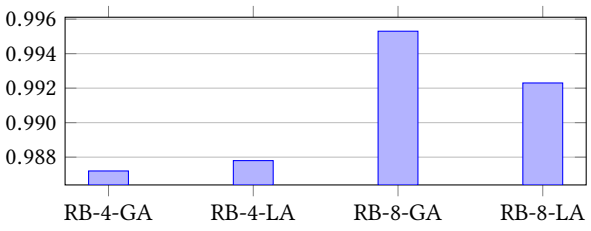


Figure 14: The Geometric mean of different checkpoint allocation algorithms, normalized to Inf.

Issue width effect. We also evaluated the effect of issue width and illustrated the results in Figure 17 which were normalized to Inf. A more aggressive processor works better as the time to re-fetch and re-execute valid instructions decreases. Moreover, we set the retire width the same as the issue width. Therefore, the time to enter the phase two mode is also reduced in a more aggressive processor.

Energy Delay Product. We evaluated the EDP of Two-Phase algorithm and compared it with an 8-checkpoint, greedy allocation architecture. The "2-phase" column in Table 4 shows the relative results. Two-Phase algorithm saves about 1% of the EDP using geometric means. The Two-Phase algorithm has to re-fetch and re-execute valid instructions during phase two recovery, thus the wasted energy increases when more instructions are re-executed, such is the case with *hmmer*. An alternative solution is to reuse the

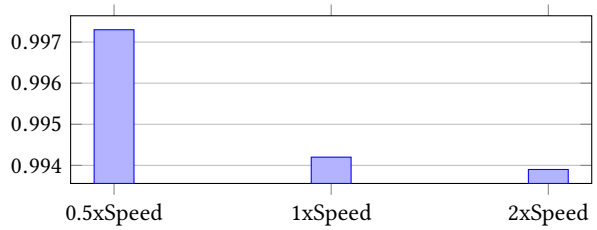


Figure 15: The Geometric mean on different memory speed, normalized to Inf.

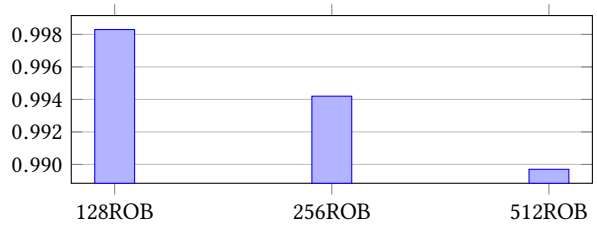


Figure 16: The Geometric mean on different rob size, normalized to Inf.

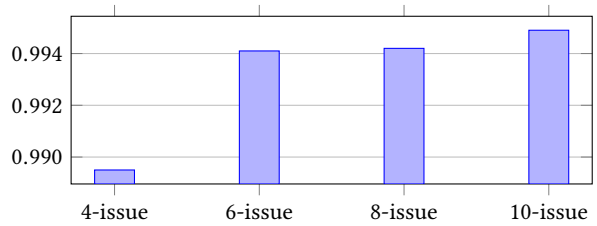


Figure 17: The Geometric mean on different issue width, normalized to Inf.

instruction results between the cache miss load and the misprediction. When the valid instructions are re-dispatched to the back-end, they are assigned the same entries in the ROB and LSQ. Therefore, the computed result is adopted if the instruction has been executed. The column of "reuse" in the Table shows the EDP result compared with the checkpoint architecture. This new design is more energy efficient.

Two-phase has to re-execute valid instructions which consumes additional energy. Table 5 shows the total energy consumption normalized to a basic recovery mechanism which does not re-execute valid instructions. It is clear that any benchmark which has many second phase recoveries (Table 1) leads to consume more energy. For example, *mcf*, *hmmer* and *astar* have the largest number of second phase recoveries. They consume the most extra energy.

7 CONCLUSION

The proposed architecture attempts to reduce the misprediction recovery penalty in two different scenarios. For the most common case, a simple and efficient basic recovery mechanism is used. For

Table 4: The EDP results of 2-phase with or without instruction reuse, normalized to a 8-checkpoint architecture

	2-phase	reuse		2-phase	reuse
perl	0.970	0.965	bzip2	0.990	0.984
gcc	0.978	0.974	mcf	1.003	0.982
gobmk	0.977	0.975	hmmer	1.061	1.030
sjeng	0.974	0.972	lib	0.982	0.982
h264ref	0.987	0.986	astar	1.011	0.992
bwaves	0.996	0.995	milc	1.001	1.000
zeusmp	0.986	0.985	gromacs	0.966	0.966
leslie3d	0.977	0.976	namd	0.993	0.993
Gems	1.000	1.000	tonto	0.989	0.989
lbm	1.000	1.000	wrf	0.998	0.998
sphinx3	0.996	0.990	gmean	0.992	0.987

Table 5: The total energy consumption of 2-phase, normalized to a basic recovery mechanism

perl	1.017	bzip2	1.029	gcc	1.018
mcf	1.071	gobmk	1.008	hmmer	1.111
sjeng	1.006	lib	1.000	h264ref	1.000
astar	1.090	bwaves	1.001	milc	1.003
zeusmp	1.000	gromacs	1.000	leslie3d	1.001
namd	1.001	Gems	1.000	tonto	1.000
lbm	1.000	wrf	1.001	sphinx3	1.010

the case in which the retire stage is blocked by a long latency operation, a *refetch-all* recovery mechanism is used to overlap branch recovery with the long latency execution. This two-Phase recovery algorithm has a performance very close to the state-of-the-art recovery mechanisms, but its implementation is significantly simpler. The structures which are used in phase one recovery can be fully reused during phase two of the recovery in an efficient way. No special tags nor checkpoints are necessary and no selective flushing is required.

We implemented and simulated our Two-Phase algorithm for single-threaded applications. We believe the algorithm can be easily implemented in a multi-core processor and is likely to yield even better outcomes. A typical multi-core processor uses a single LLC which is shared by all the cores. Therefore, the latency of LLC misses are much larger than the case with a single core processor. We already have demonstrated that Two-Phase works better when the cache miss latency is longer.

REFERENCES

[1] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2003. Checkpoint Processing and Recovery: An Efficient, Scalable Alternative to Reorder Buffers. *IEEE Micro* 23, 6 (Nov. 2003), 11–19. https://doi.org/10.1109/MM.2003.1261382

[2] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. 2003. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 423–. https://doi.org/10.1109/MICRO.2003.1253246

[3] Doug Burger and Todd M. Austin. 1997. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News* 25, 3 (June 1997), 13–25. https://doi.org/10.1145/268806.268810

[4] George Z. Chrysos and Joel S. Emer. 1998. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*. IEEE Computer Society, Washington, DC, USA, 142–153. https://doi.org/10.1145/279358.279378

[5] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004. Out-of-Order Commit Processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04)*. IEEE Computer Society, Washington, DC, USA, 48–. https://doi.org/10.1109/HPCA.2004.10008

[6] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. 2004. Toward Kilo-instruction Processors. *ACM Trans. Archit. Code Optim.* 1, 4 (Dec. 2004), 389–417. https://doi.org/10.1145/1044823.1044825

[7] Martin Dixon, Per Hammarlund, Stephan Jourdan, and Ronak Singhal. 2010. THE NEXT-GENERATION INTEL CORE MICROARCHITECTURE. *Intel Technology Journal* 14, 3 (2010).

[8] Amit Golander and Shlomo Weiss. 2008. Hiding the Misprediction Penalty of a Resource-efficient High-performance Processor. *ACM Trans. Archit. Code Optim.* 4, 4, Article 6 (Jan. 2008), 32 pages. https://doi.org/10.1145/1328195.1328201

[9] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, et al. 2001. The microarchitecture of the Pentium® 4 processor. In *Intel Technology Journal*. Citeseer.

[10] Team INRIA. [n. d.]. Branch prediction research. https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/. ([n. d.]). the 1st Championship Branch Prediction.

[11] Zhaoxiang Jin, Görkem Aşilioğlu, and Soner Önder. 2015. Mower: A New Design for Non-blocking Misprediction Recovery. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 285–294. https://doi.org/10.1145/2751205.2751228

[12] Stéphan Jourdan, Jared Stark, Tse-Hao Hsing, and Yale N. Patt. 1997. Recovery Requirements of Branch Prediction Storage Structures in the Presence of Mispredicted-path Execution. *Int. J. Parallel Program.* 25, 5 (Oct. 1997), 363–383. https://doi.org/10.1007/BF02699883

[13] R. E. Kessler. 1999. The Alpha 21264 Microprocessor. *IEEE Micro* 19, 2 (March 1999), 24–36. https://doi.org/10.1109/40.755465

[14] A.B. Kyker and D.D. Boggs. 2000. Branch recovery mechanism to reduce processor front end stall time by providing path information for both correct and incorrect instructions mixed in the instruction pool. (Feb. 15 2000). https://www.google.com/patents/US6026477 US Patent 6,026,477.

[15] J. K. F. Lee and A. J. Smith. 1984. Branch Prediction Strategies and Branch Target Buffer Design. *Computer* 17, 1 (Jan. 1984), 6–22. https://doi.org/10.1109/MC.1984.1658927

[16] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[17] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 226–237. http://dl.acm.org/citation.cfm?id=243846.243889

[18] M.S. McIlvaine, J.N. Dieffenderfer, and T.A. Sartorius. 2011. Method and apparatus for managing instruction flushing in a microprocessor's instruction pipeline. (May 24 2011). https://www.google.com/patents/US7949861 US Patent 7,949,861.

[19] Micron. [n. d.]. https://www.micron.com/resource-details/e570d65b-2664-4037-a141-620a6f2e58e7. ([n. d.]).

[20] Andreas Moshovos and Gurindar S. Sohi. 1999. Speculative Memory Cloaking and Bypassing. *Int. J. Parallel Program.* 27, 6 (Dec. 1999), 427–456. https://doi.org/10.1023/A:1018776132598

[21] Soner Önder and Rajiv Gupta. 1998. Automatic Generation of Microarchitecture Simulators. In *Proceedings of the 1998 International Conference on Computer Languages (ICCL '98)*. IEEE Computer Society, Washington, DC, USA, 80–. http://dl.acm.org/citation.cfm?id=857172.857236

[22] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.* 31, 1 (June 2003), 318–319. https://doi.org/10.1145/885651.781076

[23] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.* 10, 1 (Jan. 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[24] Andre Seznec. 2011. Storage Free Confidence Estimation for the TAGE Branch Predictor. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 443–454. http://dl.acm.org/citation.cfm?id=2014698.2014879

[25] Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*. IEEE Computer Society, Washington, DC, USA, 3–14.

[26] Kenneth C. Yeager. 1996. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro* 16, 2 (April 1996), 28–40. https://doi.org/10.1109/40.491460

[27] Peng Zhou, Soner Önder, and Steve Carr. 2005. Fast Branch Misprediction Recovery in Out-of-order Superscalar Processors. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/1088149.1088156