Appears in Intl. Symp. on High-Performance Computer Architecture (HPCA), San Francisco, CA, Feb. 2005

Checkpointed Early Load Retirement

Nevin Kırman   Meyrem Kırman   Mainak Chaudhuri*   José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

http://m3.csl.cornell.edu/

ABSTRACT

Long-latency loads are critical in today's processors due to the ever-increasing speed gap with memory. Not only do these loads block the execution of dependent instructions, they also prevent other instructions from moving through the in-order reorder buffer (ROB) and retire. As a result, the processor quickly fills up with uncommitted instructions, and computation ultimately stalls.

To attack this problem, we propose *checkpointed early load retirement*, a mechanism that combines register checkpointing and back-end—i.e., at retirement—load-value prediction. When a long-latency load hits the ROB head unresolved, the processor enters *Clear* mode by (1) taking a Checkpoint of the architectural registers, (2) supplying a Load-value prediction to consumers, and (3) EARly-retiring the long-latency load. This unclogs the ROB, thereby "clearing the way" for subsequent instructions to retire, and also allowing instructions dependent on the long-latency load to execute sooner. When the actual value returns from memory, it is compared against the prediction. A misprediction causes the processor to roll back to the checkpoint, discarding all subsequent computation. The benefits of executing in Clear mode come from providing early forward progress on correct predictions, and from warming up caches and other structures on wrong predictions.

Our evaluation shows that a Clear implementation with support for four checkpoints yields an average speedup of 1.12 for both eleven integer and eight floating-point applications (1.27 and 1.19 for five integer and five floating-point memory-bound applications, respectively), relative to a contemporary out-of-order processor with an aggressive hardware prefetcher.

1 INTRODUCTION

Modern out-of-order processors typically employ a reorder buffer (ROB) to retire instructions in program order [24]. In-order retirement enables precise bookkeeping of the architectural state, effectively making out-of-order execution transparent to the user. When, for example, an instruction raises an exception, the ROB continues to retire instructions up to the excepting one. At that point, the processor's architectural state reflects all the updates made by preceding instructions, and none of the updates made by the excepting instruction or its successors. Then, the exception handler is invoked.

In-order retirement also means that an unresolved long-latency instruction may remain at the ROB head for many cycles. This is often the case for loads that miss in the cache hierarchy, whose penalty is already severe in today's processors, and is bound to be worse in future systems due to the increasing processor-memory speed gap. These long-latency loads may hinder processor performance mainly in two ways: First, because their results are not available for many cycles, potentially long chains of dependent instructions may be blocked for a long time. Second, because instruction retirement is effectively disabled by these long-latency memory operations, executed instructions hold on to critical resources for many cycles. Upon running out of resources, the processor stops fetching new instructions and eventually stalls.

Conventional load-value prediction [3, 4, 5, 6, 16, 17, 18, 22, 28] addresses the first problem by supplying a predicted value on an unresolved load. The prediction can be provided early in the processor pipeline. Dependent instructions may then execute using this prediction. Once the value comes back from memory, it is compared against the predicted value. If they match, the instruction is deemed complete; if they do not, a replay of dependent instructions (and, possibly, all instructions after the load) takes place—this time with the right value.

In practice, however, the effectiveness of conventional load-value prediction is limited by the second problem: Indeed, because the processor must ultimately compare the loaded and the predicted values, unresolved long-latency loads continue to clog the processor at retirement. As a result, conventional load-value prediction may not be as effective with this type of loads.

Figure 1 illustrates the extent of the problem, using the baseline processor model of our experimental setup and a subset of SPEC2000 integer and floating-point applications (Section 4). It shows the fraction of the total execution time

*Now at Indian Institute of Technology, Kanpur.

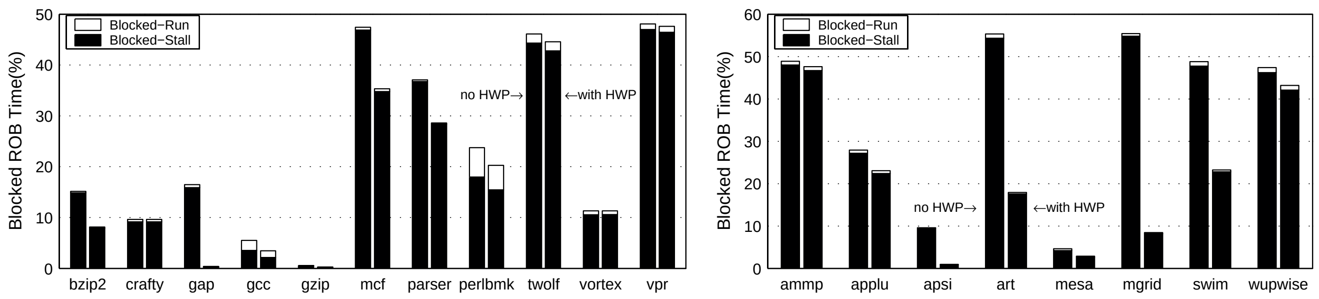


Figure 1: Percentage of total execution time that the ROB head is blocked due to an unresolved long-latency load and *(a)* the processor is still running (Blocked-Run) or *(b)* the processor is stalled as a result (Blocked-Stall). Results are shown both with and without a hardware prefetcher (HWP). Note that the y-axis scales are different for integer and floating-point applications.

that the ROB is blocked due to a long-latency L2 miss at its head. Blocked time is measured only after the access is known to have missed in L2, and is broken down into two categories, depending on whether the processor can continue issuing instructions (Blocked-Run) or it stalls due to lack of resources (Blocked-Stall). For further insight, the experiments are conducted with and without an aggressive hardware prefetcher (Section 4). (In the experiments with hardware prefetcher, blocked time is measured after a load at the ROB head is known to have missed in L2 *and* in the hardware prefetcher.)

The results reveal that, for an important number of applications, blocked ROB time accounts for a very significant fraction of the total execution time. Worse still, most of this blocked ROB time falls under Blocked-Stall category—that is, the processor is not able to do *any* work. Moreover, although the addition of a hardware prefetcher helps noticeably in a few cases, in general the problem remains. As the processor-memory speed gap widens, a solution to alleviate the problem of long-latency misses is needed.

We propose *checkpointed early load retirement*, a micro-architectural mechanism that combines register checkpointing and back-end—i.e., at retirement—load-value prediction. When a long-latency load hits the ROB head unresolved, the processor enters *Clear* mode by (1) taking a Checkpoint of the architectural registers, (2) supplying a Load-value prediction to consumers, and (3) EARly-retiring the long-latency load. This unclogs the ROB, thereby "clearing the way" for subsequent instructions to retire, and also allowing instructions dependent on the long-latency load to execute sooner. In Clear mode, subsequent long-latency loads that hit the ROB head unresolved are allowed to retire early, similarly supplying a predicted value, and selectively taking checkpoints of the architectural registers based on the confidence of the prediction. Memory updates are buffered to guarantee data integrity in case of a misprediction. When the actual values return from memory, they are compared against the predictions. A checkpoint is released only if all its early-retired loads are correctly predicted. Otherwise, the processor rolls back to that checkpoint, and all buffered memory updates are discarded.

The advantages of executing in Clear mode come in three ways: (1) On correct predictions, faster progress is achieved by enabling execution and retirement of long-latency loads and subsequent instructions. (2) On wrong predictions, later-to-be-used data and instructions are brought closer to the processor in the memory hierarchy, doing useful prefetching. (3) Different processor predictors are trained regardless of the outcome.

The paper is organized as follows: Section 2 discusses related work; Section 3 presents details of our mechanism; in Sections 4 and 5 we evaluate our proposal using a detailed simulation model and representative applications; and finally, we conclude in Section 6.

2 RELATED WORK

This work proposes the use of back-end—i.e., at retirement—load-value prediction and selective checkpointing to implement early load retirement in modern out-of-order processors. Most closely related to our work are the studies on load-value prediction, checkpointed processor architectures, and management of long-latency operations.

In conventional load-value prediction [3, 4, 5, 6, 16, 17, 18, 22, 28], which we briefly address in Section 1, the predicted load and all the subsequent instructions in program order remain in the processor, holding precious resources such as physical registers or reorder buffer entries until the load value is verified. If the load misses in all levels of the local cache hierarchy, this frequently blocks graduation, and ultimately the fetch unit, eventually bringing the processor to a stall. Our proposal solves this problem by early retiring predicted loads that take a long time to finish, and using checkpointing to roll back and refetch in case of a misprediction.

Zhou et al. [30] propose improving memory-level parallelism by speculatively pre-executing instructions with predicted values only for prefetching purposes, but without committing them—thus not requiring any validation or recovery mechanism. Register values that are not ready are predicted and supplied to the consumers in the front-end. Speculatively executed instructions remain in the issue queue for re-execution.

In the context of shared-memory multiprocessors, Chang et al. [7] and Huh et al. [12] propose decoupled coherence, by which a processor can speculatively read a matching but invalid cache line in parallel to obtaining the right permissions and value from the system. This enables overlapping the coherence protocol actions on that load with the speculative execution of the load's dependent instructions. No speculative load retirement is done; in particular, incomplete long-latency loads still block the ROB head, waiting for validation.

Bell and Lipasti [2] propose that instructions from the first few ROB slots be allowed to retire out of program order under certain conditions. Still, instructions are retired only when they have completed and are guaranteed to graduate safely.

Recently, three works explore checkpointing to overcome scalability issues of certain processor resources. Martínez et al. [19] present *Cherry*, which recycles physical registers and load/store queue entries aggressively, and uses a combination of ROB and periodic checkpointing to support precise exceptions and interrupts. Akkary et al. [1] and Cristal et al. [9, 8] present ROB-less or quasi ROB-less micro-architectures, based on a multicheckpointing mechanism. None of these works incorporates any kind of load-value prediction mechanism.

The work on Runahead execution is closest to our proposal. The concept was first proposed by Dundas and Mudge [10], and it was used to improve the data cache performance of an in-order execution core. More recently, Mutlu et al. [20] propose a Runahead architecture for out-of-order processors. The proposed architecture "nullifies" and retires a memory operation that misses in the L2 cache and remains unresolved at the time it gets to the ROB head. It also takes a checkpoint of the architectural registers, to be used to come out of Runahead mode when the memory operation completes. The instructions that depend on the nullified operation do not execute, but are nullified in turn, and hence retire quickly. Moreover, any long-latency load encountered during Runahead execution (regardless of its position in the ROB) and its dependence chain are also nullified. Other instructions execute normally, but without overwriting data in memory. When the operation completes, the processor systematically rolls back to the checkpoint and resumes conventional execution. Although execution in Runahead mode is always discarded, it effectively warms up caches and various predictors, thereby speeding up the overall execution. In contrast, our proposed design, in case of a correct load-value prediction, can continue execution without requiring a rollback. And in the case of a rollback, it can still warm up caches and predictors—albeit differently and possibly more slowly than Runahead, since dependent instructions do execute.

Lebeck et al. [15] propose a waiting instruction buffer to rid the instruction window of a long-latency load and its dependent instructions, in order to give way to subsequent independent instructions. Executed instructions are not early retired, requiring enlarged ROB and other potentially critical structures. Later, when the load value returns from memory, the instructions in the buffer are reinserted into the issue queue. Srinivasan et al. [25] improve on that concept by freeing both instruction window and register file from coping with long-latency loads and their dependent instructions. They propose a physical-to-physical register mapping, to be used when the removed slice of instructions is re-injected into the pipeline after the long-latency load completes. Their implementation is built on top of a ROB-less architecture [1]. None of these two mechanisms uses load-value prediction, and thus instructions dependent on long-latency loads cannot execute until the value returns from memory.

Karkhanis and Smith [13] analyze the reasons of performance loss due to a long-latency cache miss. They demonstrate that the structural blockage due to full ROB is a major cause. After structural limitations are removed, data dependences on the delinquent load are found to not be a significant problem. However, they show that a mispredicted branch that depends on the load may be a major cause of performance loss due to late resolution, limiting the number of useful instructions executed in the shadow of the load.

In a slightly different context, Tullsen and Brown [27] explore the performance impact of long-latency loads in an SMT processor. The authors find that it is beneficial to squash all instructions from the thread that has an outstanding long-latency load and stop fetching from that thread until the load returns. This effectively frees up critical resources, thereby allowing the other threads to make progress. In this paper, we focus on single-threaded execution, and combine (back-end) load-value prediction with selective checkpointing to realize early load retirement.

3 CHECKPOINTED EARLY LOAD RETIREMENT

The goal of checkpointed early load retirement is to tolerate long-latency loads by (1) retiring them early, so as to not block the flow of instructions through the processor, and (2) providing their dependent instructions with a predicted value. Our mechanism requires modest hardware and interferes minimally with the normal course of operation. In the following, we present our design in the context of a uniprocessor environment. Design extensions needed for multiprocessors are left for future research.

We begin by giving a high-level overview of the proposed mechanism, and then delve into implementation details.

3.1 High-level Overview

A processor executing in "conventional mode" eventually comes across a long-latency load that remains at the head of the ROB unresolved. At this point, the processor enters "Clear mode": the load is early retired, and a predicted value is supplied to its destination register. Also, the architectural registers are checkpointed. While in Clear mode, if a long-latency load arrives at the ROB head unresolved, the hardware decides whether to take a new checkpoint for this load or to use the running one. (Notice that a realistic implementation cannot afford to have more than a few checkpoints.) To make this decision, the processor consults the confidence on the prediction for this load. If the load is highly predictable (i.e., confidence is above a threshold) a new checkpoint is not allocated. On the other hand, if the load cannot be predicted with high enough confidence, a new checkpoint is allocated, provided all the checkpoints are not already exhausted.

Notice that attaching too many loads to one checkpoint increases the potential cost of a rollback, because a mispredicted value from *any* of these loads will trigger a rollback to that checkpoint. Thus, our mechanism also limits the number of loads subject to a particular checkpoint, by taking a new one when the limit is exceeded.

Naturally, only a limited number of checkpoints is supported. To overcome this limitation, once the last available checkpoint is allocated, all long-latency loads that reach the ROB head unresolved are systematically predicted and early retired, and all of them are assigned to the last checkpoint. If many such loads are encountered, the probability of suffering a rollback to that last checkpoint may be high. Yet the prefetching effect of handling these long-latency loads in Clear mode may still bring significant benefits, as our evaluation shows (Section 5). Notice that this unbounded assignment takes place only as long as there is no available checkpoint. After one becomes available, if the limit of the last one is already exceeded, or on a low-confidence load, a new checkpoint is allocated. Finally, due to structural constraints, the number of long-latency loads that can be handled in Clear mode is necessarily limited (Section 3.2).

As early-retired loads return, the processor validates the predictions against the actual returned values. A checkpoint is released when (1) it is the oldest active checkpoint, and (2a) either all its loads have been validated, or (2b) any of its loads' validation fails. In the first case, if the released checkpoint was the last one active, the processor smoothly falls back to conventional mode. On the other hand, in the last case, the processor discards the execution after the checkpoint, and uses the checkpoint to restore the architectural state.

Figure 2 shows an example of this checkpoint allocation policy. In the example, up to four checkpoints are supported, and a maximum of four loads can be assigned to a checkpoint (last checkpoint exempt). The dots on the timeline represent long-latency loads that reach the ROB head unresolved and retire early. On the first such load, regardless of its prediction confidence, always a checkpoint is taken, which initiates Clear mode. The second load, which is regarded as low-confidence, prompts allocation of a second checkpoint. The subsequent high-confidence loads are attached to this second checkpoint, until the limit of four loads per checkpoint is met. Then, on the fifth load from the last allocated checkpoint, a third checkpoint is taken regardless of prediction confidence. Later, on the next low-confidence load, a fourth and last checkpoint is allocated. Because there are no more available checkpoints, and in order to not stall execution in Clear mode at this point, subsequent loads, regardless of their prediction confidence, are assigned to this last checkpoint.

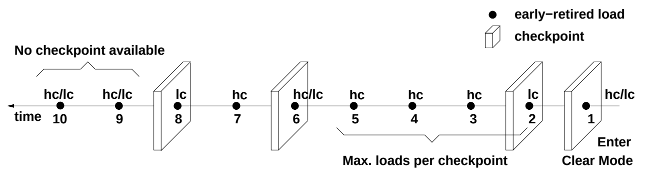


Figure 2: Checkpoint allocation example. Support of up to four checkpoints, and a maximum of four loads per checkpoint (last checkpoint exempt) is assumed. *hc* and *lc* indicate high-confidence and low-confidence loads, respectively.

3.2 Hardware Structures

Our mechanism is supported by modest hardware that can be incorporated efficiently in the processor. In this paper, we describe our proposed mechanism in the context of a contemporary micro-architecture with separate register file, ROB, and load/store queues [14, 26, 29]. Implementations over micro-architecture variations (e.g., RUU-based processor) are also possible.

In this section, we briefly comment on our mechanism's hardware additions besides checkpointing support (Section 3.3.3). These comprise a load-value predictor (*LVP*), and three FIFO queues: the *prediction queue (PQ)*, and two other queues coupled with the processor's load queue (LQ)—the *data queue (LDQ)* and the *timestamp queue (LTQ)*.

3.2.1 Load-Value Predictor

The LVP is used to generate predictions for long-latency loads at the ROB head. It is indexed by the load's instruction address. In conventional mode, the LVP is updated (and thus trained) by every load that the processor retires. In Clear mode, as a design choice, certain loads may or may not update the LVP. In our implementation, during Clear mode, all retired loads, including early-retired ones, update the LVP. However, early-retired loads do so speculatively with the predicted value. For simplicity, the hardware does not correct wrong speculative updates.

The LVP also incorporates confidence estimation [3, 16] that determines whether confidence on a given prediction is beyond a certain threshold. As already discussed, this is used in checkpoint assignment. Unlike the LVP update, early-retired loads do not update the confidence estimator.

3.2.2 Prediction Queue

The PQ is a FIFO, content-addressable (CAM) structure. Similarly to a small store queue, it sits on the processor bus and is looked up by data address. The PQ is used to remember the information associated with early-retired loads. In each entry, it stores the data address, the associated value, and the checkpoint id. The PQ has three main functions: (1) compare predicted and actual values; (2) forward predicted values to load operations as needed during Clear mode; and (3) identify the associated checkpoint on a load-value return. Entries in the PQ are allocated in order every time a new value is predicted. The size of the PQ bounds the maximum number of early-retired loads that can be handled. PQ entries are freed when their corresponding checkpoints are released. Also, once the processor exits Clear mode, the entire PQ is cleared.

3.2.3 Load Data and Timestamp Queues

The LDQ and the LTQ are two conventional FIFO RAM (but not CAM) structures that extend the processor's existing LQ (a FIFO CAM). Entries in the LQ, the LDQ, and the LTQ are allocated at rename and released at commit, in lockstep. LQ entries, as in conventional processors, store effective addresses. The LQ is used, among other important tasks (e.g., disambiguation), to match addresses with pending load instructions on a refill by the memory subsystem. LDQ entries store the values loaded by uncommitted load instructions, as returned from memory. They are used to update the LVP in order as load instructions retire (Section 3.2.1). Finally, LTQ entries store timestamps taken at the time loads are issued to the memory system. They are used to help determine if a load should be early retired, at the time it reaches the ROB head (Section 3.3.1).

3.3 Operation

We now describe in detail the different phases and mechanisms of our proposal.

3.3.1 Assessment

At the time an incomplete load reaches the ROB head, the processor decides whether to block (as in a conventional processor) or to early retire it, by determining whether this is a long-latency load. Of course, in order to be able to retire early, there must be a PQ entry available.

For this, the processor must estimate the expected response time from memory for the load. This can be accomplished in several ways.

One simple way is to count the number of cycles the load stays at the ROB head, and early retire it if it remains incomplete after a fixed number of cycles. This can be often inaccurate, but it is very easy to implement.

A more accurate estimation is possible using our LTQ support. In a typical processor, at the time the effective address of a load is calculated (shortly before the cache access is started in the processor pipeline), its LQ entry is indexed and filled with the address. In our proposed architecture, at that time, the corresponding LTQ entry is also accessed and filled with a timestamp. This timestamp corresponds, more or less, with the time at which the load issues to the memory system.

At the time an incomplete load hits the ROB head and becomes a candidate for early retirement, the processor inspects its timestamp (necessarily at the LTQ head). From the time elapsed since the load issued, the processor may determine, for example, that the load has just missed in L2. In such a case, the load may be a good candidate for early retirement. Note that, if it is determined that the load has issued only recently, the processor can continue monitoring the elapsed time until it enters L2 miss territory.

Naturally, the processor must be aware of the approximate latencies of the memory hierarchy. This characterization can be easily accomplished, for example by running a short microbenchmark [21] by the operating system at startup.

The timestamp need only be large enough to cover a worst-case round-trip to memory. In this way, no more than one wrap-around can be experienced by any one load that hits the ROB head and is still marked incomplete. Of course, we ignore the timestamp for completed loads. In any case, occasional inaccuracies in the mechanism would never affect correctness.

Overall, the processor can determine inexpensively if a particular load is eligible for early retirement.

3.3.2 Value Prediction and Speculative Retirement

At the time an unfinished load hits the ROB head, in parallel to assessing whether to early retire it, the processor makes a prediction of the load instruction's return value, by indexing the LVP with the instruction address (normally available at the ROB head, for example, to support restartable exceptions). The processor stores this prediction in a temporary buffer, to be used if the load is ultimately early retired.

If the processor decides to early retire the load, a PQ entry is allocated in FIFO order and filled as follows: The effective address is supplied by the corresponding LQ entry (necessarily at the head); the value is provided by the LVP's temporary buffer; and the id of appropriate checkpoint is recorded. Figure 3 depicts this process (LVP's temporary buffer is not shown for brevity). Finally, the predicted value is supplied to the destination register, and the load is retired.

3.3.3 Checkpointing

As already discussed in Section 3.1, the checkpoint allocation policy is based in part on the confidence of the load-value predictions.

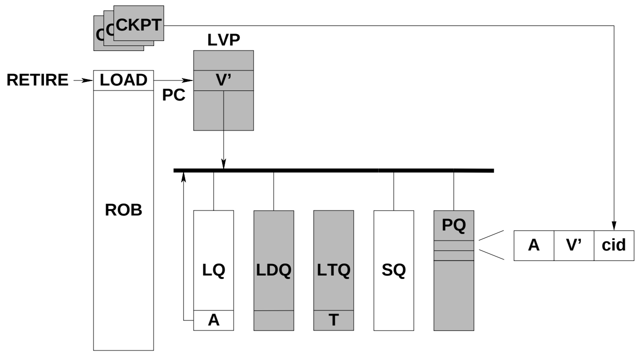


Figure 3: Simplified diagram of a Clear-enabled processor at the time a long-latency load is about to early retire (not drawn to scale). Clear hardware support is shaded. *cid*, *A*, *T*, and *V'* represent the checkpoint id, data address, timestamp, and predicted value of the load, respectively.

Checkpointing the architectural registers and restoring from a checkpoint can be done in a handful of cycles, insignificant relative to long memory latencies. A checkpoint can be acquired in several ways. One option is to keep an up-to-date shadow copy of the architectural registers during normal operation. With this option, the processor takes a checkpoint by simply freezing the shadow copy. Alternatively, the processor can explicitly copy the architectural values into the shadow register file at the time of the checkpoint. In either case, on a rollback, the architectural registers are restored using the shadow copy. A third option is to simply back up the retirement register alias table. On a rollback, the rename alias table is restored using this backup copy. This option is probably fastest, but it pins the physical registers that contain the checkpointed architectural values for the lifetime of the checkpoint.

In general, the decision of which mechanism to use depends on factors such as how many architectural registers the ISA has, how many physical registers exist in the processor, or how often checkpoints are taken. Nevertheless, our proposed mechanism can work with any of them. Further discussion of the details behind register checkpointing is outside the scope of this paper.

The following information must be kept for every checkpoint in order to monitor its state: (1) The number of early-retired loads associated with the checkpoint, and the number of such loads that have been validated, for a total of two counters. Thus, at the time a load is early retired, its checkpoint's first counter is incremented. In our implementation, bounded by the size of the PQ (48), six-bit counters are enough. (2) Whether any of the checkpoint's loads is mispredicted (one bit). This bit is set on a misprediction on any of its loads. When an early-retired load's value returns from memory, it updates the corresponding checkpoint using the checkpoint id in its PQ entry (Section 3.3.5).

In addition to the architectural registers, checkpoints of the global branch history register and return address stack (RAS) are also taken for performance reasons.

3.3.4 Speculative State

While the processor is operating in Clear mode, memory updates must be held off to prevent overwriting data that may be needed after a rollback. At the same time, these buffered updates must be available to subsequent loads. There has been a number of proposals to temporarily buffer updates following a processor checkpoint [1, 8, 19]. Conceptually speaking, they all work similarly. We explain the principle using a store buffer; in an actual implementation, any of the proposed solutions integrates well with our proposal.

While in Clear mode, store buffer entries are tagged with the ID of the checkpoint active at the time the corresponding store retires. Tagged entries are not allowed to leak out to memory. If the processor rolls back to a checkpoint—necessarily the earliest one (Section 3.3.5)—all tagged entries are gang-invalidated. If, on the other hand, the processor releases the checkpoint successfully, the tags of the entries belonging to that checkpoint are gang-cleared, and the buffered updates can lazily percolate to memory.

To support this, an additional tag field is required for each store buffer entry. Also, to hold the speculative updates, the size of the store buffer needs to be larger than usual. To have a fast but large enough buffer, a hierarchical store buffer like the one proposed in [1] can be used.

Note that, although speculative memory updates are not allowed to modify the memory state, prefetch requests for store addresses can be sent down the memory system.

In the case of filling the buffer completely with updates while in Clear mode, on a subsequent update the processor can simply stall and wait for entries to become available, as checkpoints are released and their associated updates leak to memory.

3.3.5 Verification and Release of the Checkpoints

When the memory system returns the value for an early-retired load, the processor compares the predicted and the actual values. To accomplish that, the PQ snoops the address every time a memory refill comes in, much like the LQ does.

At the time the cache controller dumps on the bus the data address and the actual value of an early-retired load, the PQ picks up the pair, finds the matching entry, and compares the predicted and the actual values. Using the checkpoint id in the PQ entry, the checkpoint's validated load counter is increased, and the misprediction bit is set if the prediction is wrong.

Every time verification occurs, the checkpoints are considered for release in order, beginning from the earliest active checkpoint in the processor. If all the loads for that checkpoint have returned from memory, and if the misprediction bit is still unset, then the checkpoint is successfully released. That is, the checkpoint is discarded, and the tags of the store buffer entries that match that checkpoint's id are gang-cleared. Then, the next active checkpoint undergoes the same procedure. If this is the last checkpoint, the processor returns to conventional mode.

If, on the other hand, the misprediction bit of the checkpoint is set, a rollback is triggered to the checkpoint. On a rollback, the pipeline is flushed, the architectural state is restored using the checkpoint, all checkpoints are released, and all tagged entries in the store buffer (regardless of id) are gang-invalidated. Then, the processor returns to conventional mode. To guarantee forward progress, the load instruction to which the processor rolls back is marked as not eligible for early retirement, in case that it misses again in the cache hierarchy.

Finally, if the checkpoint considered for release does not meet any of the above conditions, then no action is taken, and no subsequent checkpoint is considered for release until the next verification.

3.3.6 Multiple Loads

It is possible that there be a load issued subsequently to the same address as a previous early-retired load. There are two possible cases: On the one hand, there may be an intervening store, in which case the store buffer can forward the value. On the other hand, there may be no intervening store, or there may be stores with unresolved addresses that the second load bypasses speculatively. In these cases, the PQ forwards the predicted value when the load issues.

Of course, if the load is issued before the earlier load has been early retired, there is no predicted value to be forwarded (yet). However, this load has a second chance to receive the predicted value, at the time it reaches the ROB head. At that point, the PQ is inspected using the address at the LQ head, and the matching entry forwards the value. No new PQ entry is allocated for this early-retiring load.

It is also possible that there be a load to the same cache line as a predicted load, but to a different word. Again, there are two possible cases: On the one hand, there may be a buffered store, in which case the store buffer can forward the value. On the other hand, there may be no buffered store. In this case, should this second load reach the ROB head incomplete, it may be eligible for early retirement. If so, the allocation of a new PQ entry and the other actions involved are no different from what we explain above.

When a refill comes, the PQ checks for loads not only matching the refill address exactly (the critical word), but also for other loads to different words of the same cache line. On a match, the PQ "replays" the loads in this second group, so that they can pick the actual values from the cache, for comparison with the predicted ones.

3.4 Mispredictions, Exceptions, and Interrupts

In general, conventional speculative execution (e.g., branch prediction or speculative loads) is not affected by our mechanism, and mispredictions can be handled as in a conventional processor. Notice that, because a load is at the ROB head at the time early retirement takes place, it is not subject to such conventional speculative mechanisms. Furthermore, because the load must issue to the memory system before being considered for handling in Clear mode, any exception on the address calculation, address range, or page access would be caught properly.

If the processor receives an (asynchronous) interrupt while in Clear mode (e.g., an I/O request, or a quantum expiration in a multiprogrammed environment), it can handle it by stalling and waiting to revert to normal mode, either successfully or by rollback. This is also the approach in the case of operations that are irreversible or that have potential side effects, such as non-cacheable memory operations or system calls. In the case of time-sensitive interrupts, it is always possible to flush the pipeline and use the earliest checkpoint as the state to save (and return to after the interrupt). After servicing the interrupt, the processor may even inhibit Clear mode for a short period of time to guarantee progress. Of course, Clear mode could always be disabled by the programmer in time-sensitive situations, for example by using a library call.

4 EXPERIMENTAL SETUP

In this section, we discuss the simulation environment and the applications used for evaluating our Clear processor mechanism.

4.1 Simulated Architecture

We evaluate our mechanism through execution-driven simulations, using a detailed contemporary model of an out-of-order processor and its memory subsystem. The baseline processor is a four-issue dynamic superscalar running at 4GHz that has two levels of on-chip caches. The architectural parameters used are shown in Table 1. The processor has separate structures for the ROB, instruction window, and register file.

In our simulations, the latency and occupancy of the structures in the processor pipeline, caches, bus, and memory are modeled in detail. Wrong path execution, both for branch and LVP mispredictions, is also correctly simulated. We assume backup shadow ISA register files, and charge six processor cycles for taking a checkpoint of the architectural registers.

We simulate a 128-entry store queue, along the lines of the hierarchical store buffer proposed in [1]. For the sake of fairness, we use the same structure in the Baseline configuration. In the Clear configuration, PQ, LDQ, and LTQ have the same sizes as the load queue (48 entries). We use a last-value LVP, one of the simplest predictors, with 4,096 entries. The results presented in this paper use up to four checkpoints, and the checkpoint allocation threshold is set to seven loads (Section 3.3.3).

Processor	
Frequency	4GHz
Fetch/issue/commit width	4/4/6
Inst. window entries	48 Int+Mem, 32 FP
ROB entries	128
Int/FP registers	160/160
Integer FUs	4 ALU, 2/2 Mult/Div
Floating-point FUs	4 Add/Sub, 2/2 Mul/Div
Ld/St units	2/2
Ld/St queue entries	48/128
Branch penalty	16 cycles (minimum)
Max. unresolved branches	24
Branch predictor	32K-entry hybrid of GAg + bimodal 15b global history register
RAS entries	32
HW prefetcher	16-stream stride prefetcher Max. stride: 256B Operates between main memory and L2
Load-value predictor (Clear config. only)	Last-value predictor, 4K entries Confidence: 3b, Threshold: 5/7 Penalty: 2, Increment: 1
PQ, LDQ, LTQ entries (Clear config. only)	48
Memory Subsystem	
MHT entries	24 L1, 24 L2
Cache sizes	32KB L1 i-cache, 32KB L1 d-cache, 512KB L2
Cache RT (uncontended)	0.75ns L1, 4.5ns L2
Cache associativity	4-way L1, 8-way L2
Line size	64 bytes
Cache replacement policy	LRU
Cache ports	2 L1, 1 L2
System bus bandwidth	8GB/s
Memory RT (uncontended)	125ns

Table 1: Summary of the architecture modeled. In the table, MHT, RAS, and RT stand for miss handling table, return address stack, and round-trip time from the processor, respectively. Cycle counts refer to processor cycles (0.25ns).

4.2 Applications

We evaluate our mechanism using eleven SPECINT and eight SPECFP 2000 applications [11]. (At the time of this writing, our simulator cannot correctly handle the rest of SPEC applications.) We use MIPS binaries compiled at -O3 optimization level, and *ref* input sets in all cases except *ammp*. For most applications, we skip initialization and run 750 million correct, committed instructions. For the three floating-point applications for which we could not determine the initialization part (*applu*, *apsi*, *mgrid*), we skip the first 500 million instructions. Also, because *ammp* exhibits significantly different behavior over different phases [23], we simulate it to completion using the *train* input set.

5 EVALUATION

In this section we discuss our experiments and results. In Section 5.1 we comment on the speedups achieved by our proposed mechanism. In Section 5.2 we give a detailed breakdown of execution time and look at the prediction accuracy of early-retired loads. Finally, Section 5.3 presents a quantitative comparison between our mechanism and Runahead execution [20].

5.1 Performance Results

Figure 4 compares the performance attained by using an aggressive out-of-order processor without (*Baseline*) and with *Clear* support. In both cases, we also analyze the impact of adding a hardware prefetcher (*Baseline-HWP* and *Clear-HWP*, respectively). The results are represented as speedups with respect to Baseline (not shown). We also show the speedups achieved with perfect load-value prediction (*Clear-PerfectLVP*). In the rest of the section, unless noted otherwise, reported average speedups are always geometric means.

The first thing to notice is that adding a hardware prefetcher to Baseline (Baseline-HWP) attains across-the-board speedups, with a peak 2.03 for *mgrid*, and averages 1.08 and 1.16 for SPECINT and SPECFP applications, respectively. These speedups indicate that the hardware prefetcher we use in the model is effective. Yet Clear-HWP achieves a significant average speedup of 1.12 *over* Baseline-HWP for both integer and floating-point applications.

The gains obtained in memory-bound applications are even more encouraging. In this paper, we consider an application to be memory-bound if it suffers from processor stalls due to long-latency loads for over 15% of the execution time in Baseline-HWP (Figure 1). Among the integer applications, these are *mcf*, *parser*, *perlbmk*, *twolf*, and *vpr*, for which Clear-HWP yields a 1.27 average speedup relative to Baseline-HWP. Among the floating-point applications, *ammp*, *applu*, *art*, *swim*, and *wupwise* are memory-bound, and for these Clear-HWP obtains a 1.19 average speedup over Baseline-HWP.

Furthermore, when compared against the speedups obtained by Clear-PerfectLVP (100% load-value prediction accuracy), the noticeable difference conveys that there is still significant potential for further improvement, particularly in the case of memory-bound integer applications. Intuitively, this is likely to be related to the predictability of long-latency loads, which we discuss in the next section.

Overall, in light of today's widening gap between processor and memory speeds, our results indicate that early retirement of long-latency loads has great potential to yield significant performance benefits.

5.2 Breakdown of Execution Time

In this section, we analyze the sources of the performance improvements presented in Section 5.1. We break down the execution time into four categories: (1) *Conventional* execution; (2) *Clear-Correct* execution, which corresponds to computation in Clear mode for which the associated checkpoint was successfully released (and thus the computation committed); (3) *Clear-Squashed* execution, which is computation in Clear mode that is ultimately rolled back due to a load-value misprediction; and (4) *Overhead* of executing in Clear mode, mainly due to checkpoint management. Figure 5 shows this breakdown for the Clear-HWP configuration.

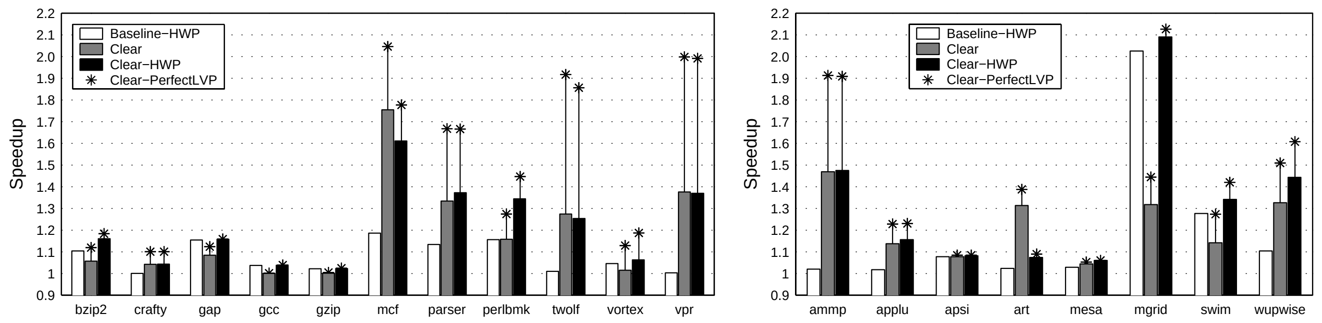


Figure 4: Performance of Baseline and Clear configurations, with and without hardware prefetcher, in integer (left) and floating-point (right) applications. An optimistic Clear configuration with perfect load-value prediction is also shown. All performance figures are speedups relative to Baseline without a hardware prefetcher (not shown).

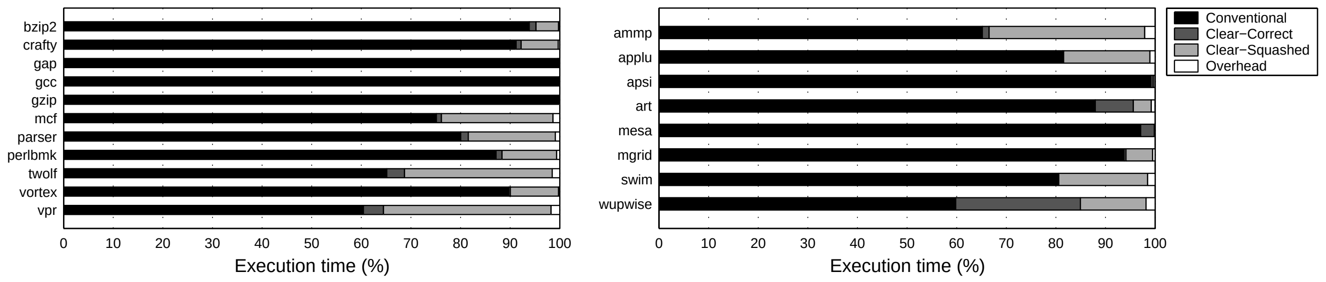


Figure 5: Breakdown of the execution time for Clear-HWP configuration in integer (left) and floating-point (right) applications.

Interestingly, the figure reveals that, for integer applications, execution in Clear mode is mostly squashed. Given the important speedups shown in Section 5.1, this implies that the performance improvements obtained in these applications are mainly due to useful prefetching performed while executing in Clear mode. On the other hand, floating-point applications show a more diverse behavior. In particular, in *art*, *mesa*, and *wupwise*, Clear-Correct constitutes a substantial part of the execution in Clear mode. This means that these applications benefit more from early retirement of (correctly predicted) long-latency loads.

To further analyze this behavior, we look at the prediction accuracy of Clear-HWP's last-value predictor (Table 1) on early-retired loads. For comparison purposes, we also examine the prediction accuracy using a more sophisticated last-four-value and stride-two-delta hybrid predictor with 2,048 entries each. (Notice that the particular organization of the load-value predictor is not fundamentally part of our proposed mechanism.) In each case, we measure prediction accuracy as actually observed in Clear-HWP executions. Because every retired load updates the predictor, this correctly captures the effect that speculative predictor updates have on subsequent predictions, including appearance of "false" long-latency accesses. We break down predictions according to the outcome (*Correct/Incorrect*) and the confidence estimation at the time of the prediction (*Confident/Unconfident*), for a total of four categories. Figure 6 shows the results.

The results show that prediction accuracy of early-retired loads is somewhat limited for most applications. Specifically, average accuracy is 28.2% and 41.8% for integer and floating-point applications, respectively. This explains the fact that execution in Clear mode is mostly squashed for many applications, particularly integer programs. In an attempt to improve this, we ran simulations using the more sophisticated last-four-value and stride-two-delta hybrid predictor mentioned before, but obtained no significant differences in predictability (third bar in Figure 6) or performance gains.

Another reason why prediction accuracy is not higher may be aliasing. Indeed, since we update the predictor with every retired load (Section 3.3.2), there is a possibility that early-retired loads are fed predictions constructed mainly from outcomes of aliased loads. One way to diminish this problem is to increase the size of the predictor. We did simulate larger predictors and found them to be similar in accuracy. Moreover, a larger predictor is not easily justifiable in terms of latency, complexity, or competitiveness against other mechanisms of similar transistor count.

Furthermore, largely as a result of the low predictabil-

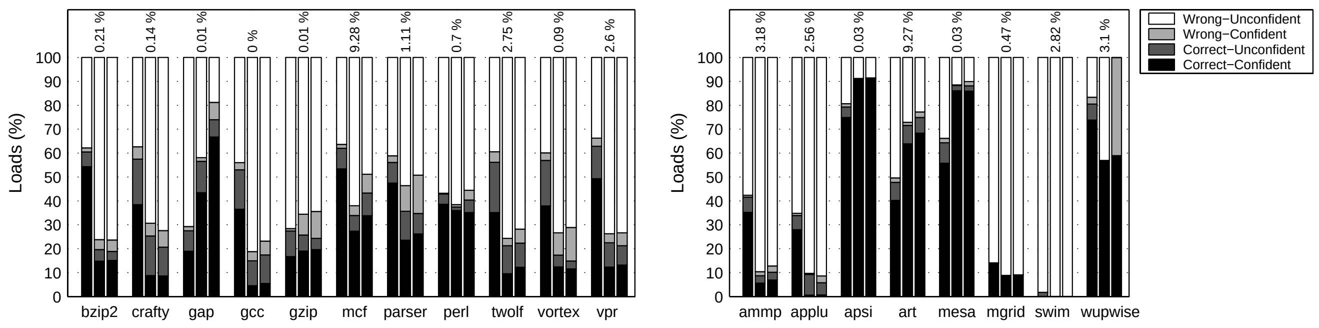


Figure 6: Prediction accuracy in the Clear-HWP configuration for all loads (first bar) and early-retired loads (second bar) using a last-value predictor (Table 1) for integer (left) and floating-point (right) applications. The percentages on top of each bar group represent the fraction of early-retired loads with respect to the total number of committed loads. The third bar shows the prediction accuracy in Clear-HWP for early-retired loads, using a more sophisticated last-four-value and stride-two-delta hybrid predictor.

ity, only 24.3% (integer) and 39.6% (floating-point) of predictions on early-retired loads are deemed confident. In our mechanism, low-confidence predictions allocate new checkpoints. Thus, these confidence levels indicate that checkpoints are expected to be in demand. More checkpoints, however, do not necessarily result in more profitable Clear-mode execution, and in fact we tried with more than four checkpoints and observed no difference in the speedups obtained.

Nevertheless, as shown, speedups are significant (Figure 4). Overall, we can identify three main factors that contribute to these performance gains: (1) successful speculative execution resulting from correct predictions; (2) useful prefetching by rolled back execution in general; and (3) training of different predictors in any case. More specifically, in the case of a rollback, instructions that are data- and control-independent of the mispredicted load value(s) can accurately prefetch data, and even dependent instructions may do useful prefetching under the right circumstances. One particular case of useful prefetching is that of instructions dependent on correctly predicted early-retired loads that are rolled back by a misprediction on a *different* load (assigned to the same or an earlier checkpoint).

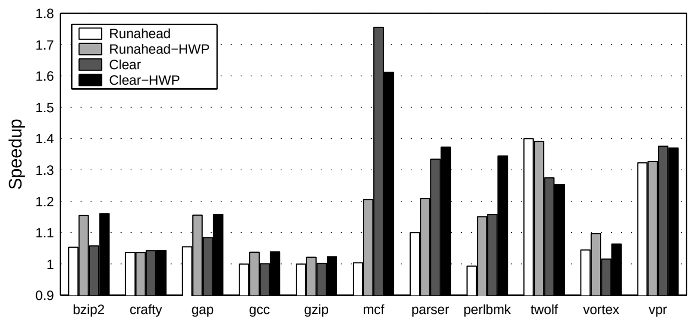
5.3 Clear vs. Runahead Execution

In this section we present a quantitative comparison between Clear and Runahead execution [20]. (For a discussion of Runahead execution, see Section 2.) Figure 7 shows speedups for Clear mode and Runahead executions, with (-*HWP*) and without hardware prefetching, relative to Baseline without hardware prefetching (not shown). Among the integer applications, Clear-HWP outperforms Runahead-HWP by a significant margin (which we define as 5% speedup or higher) in *mcf*, *parser* and *perlbmk*. In fact, in *mcf* and *parser*, Clear alone (without a hardware prefetcher) outperforms Runahead-HWP by such a margin. On the other hand, Runahead-HWP outperforms Clear-HWP in this way only in *twolf*.

Among the floating-point applications, Clear-HWP outperforms Runahead-HWP significantly in two applications, *ammp* and *wupwise*. In the remaining applications, both techniques perform similarly on average. Overall, in our experiments, our mechanism significantly outperforms Runahead in five occasions, while Runahead beats Clear-mode execution by a significant margin in only one case.

There are several properties unique to executing in Clear mode that help outperform Runahead in some applications. Here we list a few: (1) Rollbacks may be avoided if early-retired loads are predicted correctly. Runahead systematically rolls back to the checkpoint after the first early-retired load completes. (2) On a misprediction, a processor running in Clear mode may still benefit from prefetching by executing instruction chains dependent on early-retired loads, particularly correctly predicted ones. Runahead generally nullifies early-retired loads and their dependent instruction chains. (3) A particular case of (2) is that of branches dependent on early-retired loads. While both Runahead and Clear mechanisms support branch prediction for those, only in the latter can they resolve, using the predicted load value. Whenever the predicted value is correct—or even if incorrect but sufficient to resolve the branch correctly, our mechanism can help redirect a mispredicted branch and resume execution along the correct path. Interestingly, Karkhanis and Smith [13] show that the number of branches dependent on long-latency loads can be significant in the SPEC2000 integer applications. (4) A store whose address depends on a mispredicted early-retired load can still successfully forward its value to a subsequent load whose address depends on that same mispredicted load, whenever both the dependent store and load resolve to the same (wrong) address.

Overall, these properties help the Clear implementation outperform Runahead in the five applications mentioned above. On the other hand, Runahead's best advantage over



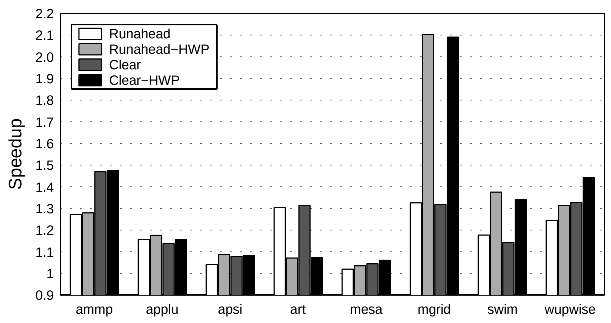


Figure 7: Performance of Clear and Runahead execution, with (*-HWP*) and without hardware prefetching, for integer (left) and floating-point (right) applications. Speedups are relative to Baseline without hardware prefetching (not shown). Note the different scales on the Y axis for integer and floating-point applications.

our proposed mechanism is its ability to race deeper into the code after the checkpoint, by systematically nullifying long-latency loads and their dependent instructions. This becomes evident in Runahead's success over Clear-mode execution in *twolf*.

We notice that, in a few applications, Runahead obtains speedups that are somewhat different from the numbers reported in [20]. Even though our implementation of Runahead execution is conceptually identical to what is proposed in [20], our base architecture (on which we build Runahead) is different from what is used in [20]. In particular, our MIPS ISA-based load-store architecture offers a larger number of logical registers compared to the x86 architecture used in [20], which typically results in less spill code. Also, in place of a trace cache, we model a conventional instruction cache.

6 CONCLUSIONS

We have proposed *checkpointed early load retirement*, a micro-architectural mechanism based on selective processor checkpointing and back-end (i.e., at retirement) load-value prediction. When a long-latency load reaches the ROB head unresolved, the processor (1) takes a checkpoint of the architectural registers, (2) supplies a load-value prediction to consumers, and (3) early-retires the long-latency load. This we call *Clear* mode of execution. It allows instruction retirement to resume and dependent instructions to execute sooner, thereby reducing processor stalls. When the long-latency load completes, the returned value is compared against the prediction. On a correct prediction, execution in Clear mode is deemed correct and the checkpoint is released. On a misprediction, execution in Clear mode is rolled back and the processor reverts to the checkpointed state. Nevertheless, in that case, instructions executed in Clear mode may have a prefetching effect on instructions and data. Finally, regardless of the prediction outcome, the different predictors in the processor are trained by executing in Clear mode.

We have proposed a general solution that supports multiple checkpoints and multiple early-retired loads per checkpoint. Our mechanism requires modest hardware additions on top of a conventional ROB-based processor. In particular, it does not require upsizing the ROB, the register file, or the instruction queues.

Detailed simulations of our proposed mechanism reveal that, compared to a state-of-the-art baseline architecture with an aggressive hardware prefetcher, our mechanism achieves important speedups for a set of integer and floating-point applications. We find that the prefetching benefits of squashed execution in Clear mode are important for many applications, for which predictability of early-retired loads is limited. When compared against Mutlu et al.'s Runahead execution [20], our proposed mechanism significantly outperforms it (5% speedup or higher) in five applications, vs. only one application for which Runahead works significantly better (5% speedup or higher) than our mechanism.

Overall, our study concludes that our proposed mechanism constitutes an effective way to confront the growing disparity of processor and memory speeds. We are extending our work in various ways, including (1) integration with conventional (front-end) load-value prediction, and (2) integration with other checkpoint-based processor architectures of complementary objectives, such as Cherry's support for more in-flight instructions through aggressive resource recycling [19].

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback. This work was supported in part by NSF grant CCF-0429922 and gifts from Intel. Meyrem Kırman was supported in part by a McMullen fellowship.

REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction

window processors. In *International Symposium on Microarchitecture*, pages 423–434, San Diego, CA, December 2003.

[2] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *International Symposium on Performance Analysis of Systems and Software*, Austin, TX, March 2004.

[3] M. Burtscher and B. G. Zorn. Hybrid load-value predictors. *IEEE Transactions on Computers*, 51(7):759–774, July 2002.

[4] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction-Level Parallelism*, 1, March 1999.

[5] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[6] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *International Symposium on Computer Architecture*, pages 64–74, Atlanta, GA, May 1999.

[7] J. Chang, J. Huh, R. Desikan, D. Burger, and G. S. Sohi. Using coherent value speculation to improve multiprocessor performance. In *First Value-Prediction Workshop*, San Diego, CA, June 2003.

[8] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *International Symposium on High-Performance Computer Architecture*, pages 48–59, Madrid, Spain, February 2004.

[9] A. Cristal, M. Valero, J.-L. Llosa, and A. González. Large virtual ROBs by processor checkpointing. Technical Report UPC-DAC-2002-39, Universitat Politecnica de Catalunya, July 2002.

[10] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *International Conference on Supercomputing*, pages 68–75, Vienna, Austria, July 1997.

[11] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[12] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 2004.

[13] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*, Anchorage, Alaska, May 2002.

[14] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 9(2):24–36, March 1999.

[15] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *International Symposium on Computer Architecture*, pages 59–70, Anchorage, AK, May 2002.

[16] M. H. Lipasti. *Value Locality and Speculative Execution*. Ph.D. dissertation, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, May 1997.

[17] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 1996.

[18] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Cambridge, MA, October 1996.

[19] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *International Symposium on Microarchitecture*, Istanbul, Turkey, November 2002.

[20] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *International Symposium on High-Performance Computer Architecture*, pages 129–140, Anaheim, CA, February 2003.

[21] R. H. Saavedra-Barrera. *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*. Ph.D. dissertation, University of California, Berkeley, May 1992.

[22] Y. Sazeides and J. E. Smith. The predictability of data values. In *International Symposium on Microarchitecture*, pages 248–258, Research Triangle Park, NC, December 1997.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, San Jose, CA, October 2002.

[24] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[25] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Boston, MA, October 2004.

[26] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[27] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *International Symposium on Microarchitecture*, pages 318–327, Austin, TX, December 2001.

[28] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *International Symposium on Microarchitecture*, pages 281–290, Research Triangle Park, NC, December 1997.

[29] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 6(2):28–40, April 1996.

[30] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *International Conference on Supercomputing*, pages 326–335, San Francisco, CA, June 2003.