Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors*

José F. Martínez Jose Renau[†] Michael C. Huang[‡] Milos Prvulovic[†] Josep Torrellas[†]

Computer Systems Laboratory, Cornell University
martinez@csl.cornell.edu

[†]Dept. of Computer Science, University of Illinois at Urbana-Champaign
{renau,prvulovi,torrellas}@cs.uiuc.edu

[‡]Dept. of Electrical and Computer Engineering, University of Rochester
michael.huang@ece.rochester.edu

ABSTRACT

This paper presents *CHeckpointed Early Resource RecYcling* (*Cherry*), a hybrid mode of execution based on ROB and checkpointing that decouples resource recycling and instruction retirement. Resources are recycled early, resulting in a more efficient utilization. Cherry relies on state checkpointing and rollback to service exceptions for instructions whose resources have been recycled. Cherry leverages the ROB to (1) not require in-order execution as a fallback mechanism, (2) allow memory replay traps and branch mispredictions without rolling back to the Cherry checkpoint, and (3) quickly fall back to conventional out-of-order execution without rolling back to the checkpoint or flushing the pipeline.

We present a Cherry implementation with early recycling at three different points of the execution engine: the load queue, the store queue, and the register file. We report average speedups of 1.06 and 1.26 in SPECint and SPECfp applications, respectively, relative to an aggressive conventional architecture. We also describe how Cherry and speculative multithreading can be combined and complement each other.

1 INTRODUCTION

Modern out-of-order processors typically employ a reorder buffer (ROB) to retire instructions in order [18]. In-order retirement enables precise bookkeeping of the architectural state, while 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.

One disadvantage of typical ROB implementations is that individual instructions hold most of the resources that they use until they retire. Examples of such resources are load/store queue entries and physical registers [1, 6, 21, 23]. As a result, an instruction that completes early holds on to these resources for a long time, even if it does not need them anymore. Tying up unneeded resources limits performance, as new instructions may find nothing left to allocate.

To tackle this problem, we propose *CHeckpointed Early Resource RecYcling* (*Cherry*). Cherry is a mode of execution that decouples the recycling of the resources used by an instruction and the retirement of the instruction. Resources are released early and gradually and, as a result, they are utilized more efficiently. For a processor with a given level of resources, Cherry's early recycling can boost the performance; alternatively, Cherry can deliver a given level of performance with fewer resources.

While Cherry uses the ROB, it also relies on state checkpointing to roll back to a correct architectural state when exceptions arise for instructions whose resources have already been recycled. When this happens, the processor re-executes from the checkpoint in conventional *out-of-order* mode (non-Cherry mode). At the time the exception re-occurs, the processor handles it precisely. Thus, Cherry supports precise exceptions. Moreover, Cherry uses the cache hierarchy to buffer memory system updates that may have to be undone in case of a rollback; this allows much longer checkpoint intervals than a mechanism limited to a write buffer.

At the same time, Cherry leverages the ROB to (1) not require in-order execution as a fallback mechanism, (2) allow memory replay traps and branch mispredictions without rolling back to the Cherry checkpoint, and (3) quickly fall back to conventional out-of-order execution without rolling back to the checkpoint or even flushing the pipeline.

To illustrate the potential of Cherry, we present an implementation on a processor with separate structures for the instruction window, ROB, and register file. We perform early recycling at three key points of the execution engine: the load queue, the store queue, and the register file. To our knowledge, this is the first proposal for early recycling of load/store queue entries in processors with load speculation and replay traps. Overall, this Cherry implementation results in average speedups of 1.06 for SPECint and 1.26 for SPECfp applications, relative to an aggressive conventional architecture with an equal amount of such resources.

Finally, we discuss how to combine Cherry and Speculative Multithreading (SM) [4, 9, 14, 19, 20]. These two checkpoint-based techniques complement each other: while Cherry uses potentially unsafe resource recycling to enhance instruction overlap *within* a thread, SM uses potentially unsafe parallel execution to enhance instruction overlap *across* threads. We demonstrate how a combined scheme reuses much of the hardware required by either technique.

This paper is organized as follows: Section 2 describes Cherry in detail; Section 3 explains the three recycling mechanisms used in this work; Section 4 presents our setup to evaluate Cherry; Section 5 shows the evaluation results; Section 6 presents the integration of Cherry and SM; and Section 7 discusses related work.

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM, Intel, and Hewlett-Packard.

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

2 CHERRY: CHECKPOINTED EARLY RESOURCE RECYCLING

The idea behind Cherry is to decouple the recycling of the resources consumed by an instruction and the retirement of the instruction. A Cherry-enabled processor recycles resources as soon as they become unnecessary in the normal course of operation. As a result, resources are utilized more efficiently. Early resource recycling, however, can make it hard for a processor to achieve a consistent architectural state if needed. Consequently, before a processor enters Cherry mode, it makes a checkpoint of its architectural registers in hardware (Section 2.1). This checkpoint may be used to roll back to a consistent state if necessary.

There are a number of events whose handling requires gathering a precise image of the architectural state. For the most part, these events are memory replay traps, branch mispredictions, exceptions, and interrupts. We can divide these events into two groups:

The first group consists of memory replay traps and branch mispredictions. A memory replay trap occurs when a load is found to have issued to memory out of order with respect to an older memory operation that overlaps [1]. When the event is identified, the offending load and all younger instructions are re-executed (Section 3.1.1). A branch misprediction squashes all instructions younger than the branch instruction, after which the processor initiates the fetching of new instructions from the correct path.

The second group of events comprises exceptions and interrupts. In this paper we use the term *exception* to refer to any *synchronous* event that requires the precise architectural state at a particular instruction, such as a division by zero or a page fault. In contrast, we use *interrupt* to mean *asynchronous* events, such as I/O or timer interrupts, which are not directly associated with any particular instruction.

The key aspect that differentiates these two groups is that, while memory replay traps and branch mispredictions are a common, direct consequence of ordinary speculative execution in an aggressive out-of-order processor, interrupts and exceptions are extraordinary events that occur relatively infrequently.

As a result, the philosophy of Cherry is to allow early recycling of resources *only* when they are not needed to support (the relatively common) memory replay traps and branch mispredictions. However, recycled resources may be needed to service extraordinary events, in which case the processor restores the checkpointed state and restarts execution from there (Section 2.3.3).

To restrict resource recycling in this way, we identify a ROB entry as the *Point of No Return* (*PNR*). The PNR corresponds to the oldest instruction that can still suffer a memory replay trap or a branch misprediction (Figure 1). Early resource recycling is allowed only for instructions older than the PNR.

Instructions that are no older than the PNR are called *reversible*. In these instructions, when memory replay traps, branch mispredictions, or exceptions occur, they are handled as in a conventional out-of-order processor. It is never necessary to roll back to the checkpointed state. In particular, exceptions raised by reversible instructions are precise.

Instructions that are older than the PNR are called *irreversible*. Such instructions may or may not have completed their execution. However, some of them may have released their resources. In the event that an irreversible instruction raises an exception, the processor has to roll back to the checkpointed state. Then, the processor executes in conventional *out-of-order* mode (non-Cherry or normal mode) until the exception re-occurs. When the exception re-occurs, it is handled in a *precise* manner as in a conventional processor. Then, the processor can return to Cherry mode if desired (Section 2.3.3).

As for interrupts, because of their asynchronous nature, they are always handled without any rollback. Specifically, processor execution seamlessly falls back to non-Cherry mode (Section 2.1). Then, the interrupt is processed. After that, the processor can return to Cherry mode.

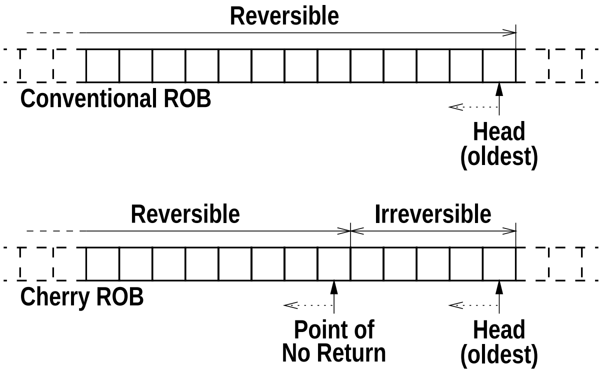

Figure 1: Conventional ROB and Cherry ROB with the Point of No Return (PNR). We assume a circular ROB implementation with Head and Tail pointers [18].

The position of the PNR depends on the particular implementation and the types of resources recycled. Conservatively, the PNR can be set to the oldest of (1) the oldest unresolved branch instruction (U_B), and (2) the oldest memory instruction whose address is still unresolved (U_M). Instructions older than oldest(U_B, U_M) are not subject to reply traps or squashing due to branch misprediction. If we define U_L and U_S as the oldest load and store instruction, respectively, whose address is unresolved, the PNR expression becomes oldest(U_B, U_L, U_S). In practice, a more aggressive definition is possible in some implementations (Section 3).

In the rest of this section, we describe Cherry as follows: first, we introduce the basic operation under Cherry mode; then, we describe needed cache hierarchy support; next, we address events that cause the squash and re-execution of instructions; finally, we examine an important Cherry parameter.

2.1 Basic Operation under Cherry Mode

Before a processor can enter Cherry mode, a checkpoint of the architectural register state has to be made. A simple support for checkpointing includes a backup register file to keep the checkpointed register state and a retirement map at the head of the ROB. Of course, other designs are possible, including some without a retirement map [23].

With this support, creating a checkpoint involves copying the architectural registers pointed to by the retirement map to the backup register file, either eagerly or lazily. If it is done eagerly, the copying can be done in a series of bursts. For example, if the hardware supports four data transfers per cycle, 32 architectural values can be backed up in eight processor cycles. Note that the backup registers are not designed to be accessed by conventional operations and, therefore, they are simpler and take less silicon than the main physical registers. If the copying is done lazily, the physical registers pointed to by the retirement map are simply tagged. Later, each of them is backed up before it is overwritten.

While the processor is in Cherry mode, the PNR races ahead of the ROB head (Figure 1), and early recycling takes place in the irreversible set of instructions. As in non-Cherry mode, the retirement map is updated as usual as instructions retire. Note, however, that the retirement map may point to registers that have already been recycled and used by other instructions. Consequently, the true architectural state is unavailable—but reconstructible, as we explain below.

Under Cherry mode, the processor boosts the IPC through more efficient resource utilization. However, the processor is subject to exceptions that may cause a costly rollback to the checkpoint. Consequently, we do not keep the processor in Cherry mode indefinitely. Instead, at some point, the processor falls back to non-Cherry mode.

This can be accomplished by simply freezing the PNR. Once all instructions in the irreversible set have retired, and thus the ROB head has caught up with the PNR, the retirement map reflects the true architectural state. By this time, all the resources that were recycled early would have been recycled in non-Cherry mode too. This *collapse step* allows the processor to fall back to non-Cherry mode smoothly. Overall, the checkpoint creation, early recycling, and collapse step is called a *Cherry cycle*.

Cherry can be used in two ways. One way is to enter Cherry mode only as needed, for example when the utilization of one of the resources (physical register file, load queue, etc.) reaches a certain threshold. This situation may be caused by an event such as a long-latency cache miss. Once the pressure on the resources falls below a second threshold, the processor returns to non-Cherry mode. We call this use *on-demand Cherry*.

Another way is to run in Cherry mode continuously. In this case, the processor kepts executing in Cherry mode irrespective of the regime of execution, and early recycling takes place all the time. However, from time to time, the processor needs to take a new checkpoint in order to limit the penalty of an exception in an irreversible instruction. To generate a new checkpoint, we simply freeze the PNR as explained before. Once the collapse step is completed, a new checkpoint is made, and a new Cherry cycle starts. We call this use *rolling Cherry*.

2.2 Cache Hierarchy Support

While in Cherry mode, the memory system receives updates that have to be discarded if the processor state is rolled back to the checkpoint. To support long Cherry cycles, we must allow such updates to overflow beyond the processor buffers. To make this possible, we keep all these processor updates within the local cache hierarchy, disallowing the spill of any such updates into main memory. Furthermore, we add one *Volatile* bit in each line of the local cache hierarchy to mark the updated lines.

Writes in Cherry mode set the Volatile bit of the cache line that they update. Reads, however, are handled as in non-Cherry mode.[1] Cache lines with the Volatile bit set may not be displaced beyond the outermost level of the local cache hierarchy, e.g. L2 in a two-level structure. Furthermore, upon a write to a cached line that is marked dirty but not Volatile, the original contents of the line are written back to the next level of the memory hierarchy, to enable recovery in case of a rollback. The cache line is then updated, and remains in state dirty (and now Volatile) in the cache.

If the processor needs to roll back to the checkpoint while in Cherry mode, all cache lines marked Volatile in its local cache hierarchy are gang-invalidated as part of the rollback mechanism. Moreover, all Volatile bits are gang-cleared. On the other hand, if the processor successfully falls back to non-Cherry mode, or if it creates a new checkpoint while in rolling Cherry, we simply gang-clear the Volatile bits in the local cache hierarchy.

These gang-clear and gang-invalidation operations can be done in a handful of cycles using inexpensive custom circuitry. Figure 2 shows a bit cell that implements one Volatile bit. It consists of a standard 6-T SRAM cell with one additional transistor for gang-clear (inside the dashed circle). Assuming a 0.18μm TSMC process, and using SPICE to extract the capacitance of a line that would gang-clear 8Kbits (one bit per 64B cache line in a 512KB cache), we estimate that the gang-clear operation takes 6-10 FO4s [13]. If the delay of a processor pipeline stage is about 6-8 FO4s, gang-clearing can be performed in about two processor cycles. Gang-invalidation simply invalidates all lines whose Volatile bit is set (by gang-clearing their Valid bits).

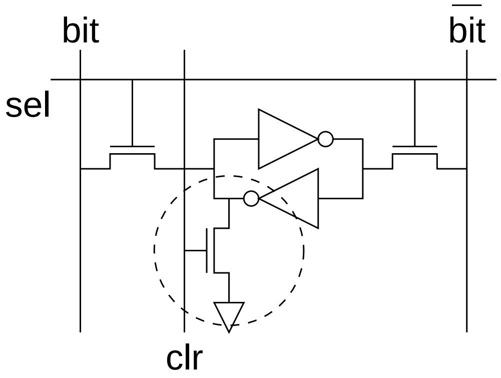

Figure 2: Example of the implementation of a Volatile bit with a typical 6-T SRAM cell and an additional transistor for gang-clear (inside the dashed circle).

Finally, we consider the case of a cache miss in Cherry mode that cannot be serviced due to lack of evictable cache lines (all lines in the set are marked Volatile). In general, if no space can be allocated, the application must roll back to the checkpoint. To prevent this from hapenning too often, one may bound the length of a Cherry cycle, after which a new checkpoint is created. However, a more flexible solution is to include a fully associative victim cache in the local cache hierarchy, that accommodates evicted lines marked Volatile. When the number of Volatile lines in the victim cache exceeds a certain threshold, an interrupt-like signal is sent to the processor. As with true interrupts (Section 2.3.2), the processor proceeds with a collapse step and, once in non-Cherry mode, gang-clears all Volatile bits. Then, a new Cherry cycle may begin.

2.3 Squash and Re-Execution of Instructions

The main events that cause the squash and possible re-execution of in-flight instructions are memory replay traps, branch mispredictions, interrupts, and exceptions. We consider how each one is handled in Cherry mode.

2.3.1 Memory Replay Traps and Branch Mispredictions

Only instructions in the reversible set (Figure 1) may be squashed due to memory replay traps or branch mispredictions. Since resources have not yet been recycled in the reversible set, these events can be handled in Cherry mode conventionally. Specifically, in a replay trap, the offending load and all the subsequent instructions are replayed; in a branch misprediction, the instructions in the wrong path are squashed and the correct path is fetched.

2.3.2 Interrupts

Upon receiving an interrupt while in Cherry mode, the hardware automatically initiates the transition to non-Cherry mode by entering a collapse step. Once non-Cherry mode is reached, the processor handles the interrupt as usual.

[1] Read misses that find the requested line marked Volatile in a lower level of the cache hierarchy also set (inherit) the Volatile bit. This is done to ensure that the lines with updates are correctly identified in a rollback.

Note that Cherry handles interrupts *without rolling back to the checkpoint*. The only difference with respect to a conventional processor is that the interrupt may have a slightly higher response time. Depending on the application, we estimate the increase in the response time to range from tens to a few hundred nanoseconds. Such an increase is tolerable for typical asynchronous interrupts. In the unlikely scenario that this extra latency is not acceptable, the simplest solution is to disable Cherry.

2.3.3 Exceptions

A processor running in Cherry mode handles all exceptions precisely. An exception is processed differently depending on whether it occurs on a reversible or an irreversible instruction (Figure 1). When it occurs on a reversible one, the corresponding ROB entry is marked. If the instruction is squashed before the PNR gets to it (e.g. it is in the wrong path of a branch), the (false) exception will have no bearing on Cherry. If, instead, the PNR reaches that ROB entry while it is still marked, the processor proceeds to exit Cherry mode (Section 2.1): the PNR is frozen and, as execution proceeds, the ROB head eventually catches up with the PNR. At that point, the processor is back to non-Cherry mode and, since the excepting instruction is at the ROB head, the appropriate handler is invoked.

If the exception occurs on an irreversible instruction, the hardware automatically rolls back to the checkpointed state and restarts execution from there in non-Cherry mode. Rolling back to the checkpointed state involves aborting any outstanding memory operations, gang-invalidating all cache lines marked Volatile, gang-clearing all Volatile bits, restoring the backup register file, and starting to fetch instructions from the checkpoint. The processor executes in conventional out-of-order mode (non-Cherry mode) until the exception re-occurs. At that point, the exception is processed normally, after which the processor can re-enter Cherry mode.

It is possible that the exception does not re-occur. This may be the case, for example, for page faults in a shared-memory multiprocessor environment. Consequently, we limit the number of instructions that the processor executes in non-Cherry mode before returning to Cherry mode. One could remember the instruction that caused the exception and only re-execute in non-Cherry mode until such instruction retires. However, a simpler, conservative solution that we use is to remain in non-Cherry mode until we retire the number of instructions that are executed in a Cherry cycle. Section 2.4 discusses the optimal size of a Cherry cycle.

2.3.4 OS and Multiprogramming Issues

Given that the operating system performs I/O mapped updates and other hard-to-undo operations, it is advisable not to use Cherry mode while in the OS kernel. Consequently, system calls and other entries to the kernel automatically exit Cherry mode.

However, Cherry blends well with context switching and multiprogramming. If a timer interrupt mandates a context switch for a process, the processor bails out of Cherry mode as described in Section 2.3.2, after which the resident process can be preempted safely. If it is the process who yields the CPU (e.g. due to a blocking semaphore), the system call itself exits Cherry mode, as described above. In no case is a rollback to the checkpoint necessary.

2.4 Optimal Size of a Cherry Cycle

The size of a Cherry cycle is crucial to the performance of Cherry. In what follows, we denote by T_c the duration of a Cherry cycle ignoring any Cherry overheads. For Cherry to work well, T_c has to be within a range. If T_c is too short, performance is hurt by the overhead of the checkpointing and the collapse step. If, instead, T_c is too long, both the probability of suffering an exception within a Cherry cycle and the cost of a rollback are high. The optimal T_c is found somewhere in between these two opposing conditions. We now show how to find the optimal T_c. For simplicity, in the following discussion we assume that the IPC in Cherry and non-Cherry mode stays constant at $s \cdot$IPC and IPC, respectively, where s denotes the average overhead-free speedup delivered by the Cherry mode.

We can express the per-Cherry-cycle overhead T_o of running in Cherry mode as:

$$T_o = c_k + p_e c_e \tag{1}$$

where c_k is the overhead caused by checkpointing and by the reduced performance experienced in the subsequent collapse step, p_e is the probability of suffering a rollback-causing exception in a Cherry cycle, and c_e is the cost of suffering such an exception. If exceptions occur every T_e cycles, with $T_c < T_e$, we can rewrite:

$$p_e = \frac{T_c}{T_e} \tag{2}$$

Notice that the expression for p_e is conservative, since it assumes that all exceptions cause rollbacks. In reality, only exceptions triggered by instructions in the irreversible set cause the processor to roll back, and thus the actual p_e would be lower.

To calculate the cost of suffering such an exception, we assume that when exceptions arrive, they do so half way into a Cherry cycle. In this case, the cost consists of re-executing half Cherry cycle at non-Cherry speed, plus the incremental overhead of executing (for the first time) another half Cherry cycle at non-Cherry speed rather than at Cherry speed. Recall that, after suffering an exception, we execute the instructions of one full Cherry cycle in non-Cherry mode (Section 2.3.3). Consequently:

$$c_e = s\frac{T_c}{2} + (s-1)\frac{T_c}{2} \tag{3}$$

The optimal T_c is the one that minimizes T_o/T_c. Substituting Equation 2 and Equation 3 in Equation 1, and dividing by T_c yields:

$$\frac{T_o}{T_c} = \frac{c_k}{T_c} + \left(s - \frac{1}{2}\right)\frac{T_c}{T_e} \tag{4}$$

This expression finds a minimum in:

$$T_c = \sqrt{\frac{c_k T_e}{s - \frac{1}{2}}} \tag{5}$$

Figure 3 plots the relative overhead T_o/T_c against the duration T_c of an overhead-free Cherry cycle (Equation 4). For that experiment, we borrow from our evaluation section (Section 5): 3.2GHz processor, $c_k = 18.75$ns (60 cycles), and $s = 1.06$. Then, we plot curves for duration of interval between exceptions T_e ranging from 200μs to 1000μs. The minimum in each curve yields the optimal T_c (Equation 5). As we can see from the figure, for the parameters assumed, the optimal T_c hovers around a few microseconds.

3 EARLY RESOURCE RECYCLING

To illustrate Cherry, we implement early recycling in the load/store unit (Section 3.1) and register file (Section 3.2). Early recycling could be applied to other resources as well.

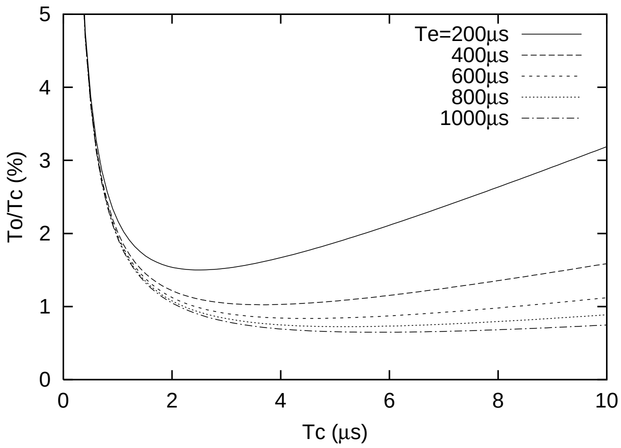

Figure 3: Example of Cherry overheads for different intervals between exceptions (T_e) and overhead-free Cherry cycle durations (T_c).

3.1 Load/Store Unit

Typical load/store units comprise one reorder queue for loads and one for stores [1, 21]. Either reorder queue may become a performance bottleneck if it fills up. In this section we first discuss a conventional design of the queues, and then we propose a new mechanism for early recycling of load/store queue entries.

3.1.1 Conventional Design

The processor assigns a Load Queue (LQ) entry to every load instruction in program order, as the instruction undergoes renaming. The entry initially contains the destination register. As the load executes, it fills its LQ entry with the appropriate physical address and issues the memory access. When the data are obtained from the memory system, they are passed to the destination register. Finally, when the load finishes and reaches the head of the ROB, the load instruction retires. At that point, the LQ entry is recycled.

Similarly, the processor assigns Store Queue (SQ) entries to every store instruction in program order at the renaming stage. As the store executes, it generates the physical address and the data value, which are stored in the corresponding SQ entry. An entry whose address and data are still unknown is said to be *empty*. When both address and data are known, and the corresponding store instruction reaches the head of the ROB, the update is sent to the memory system. At that point, the store retires and the SQ entry is recycled.

Address Disambiguation and Load-Load Replay

At the time a load generates its address, a disambiguation step is performed by comparing its physical address against that of older SQ entries. If a fully overlapping entry is found, and the data in the SQ entry are ready, the data are forwarded to the load directly. However, if the accesses fully overlap but the data are still missing, or if the accesses are only partially overlapping, the load is rejected, to be dispatched again after a number of cycles. Finally, if no overlapping store exists in the SQ that is older than the load, the load requests the data from memory at once.

The physical address is also compared against newer LQ entries. If an overlapping entry is found, the newer load and all its subsequent instructions are replayed, to eliminate the chance of an intervening store by another device causing an inconsistency.

This last event, called *load-load replay trap*, is meaningful only in environments where more than one device can be accessing the same memory region simultaneously, as in multiprocessors. In uniprocessor environments, such a situation could potentially be caused by processor and DMA accesses. However, in practice, it does not occur: the operating system ensures mutual exclusion of processor and DMA accesses by locking memory pages as needed. Consequently, load-load replay support is typically not necessary in uniprocessors.

Figure 4*(a)* shows an example of a load address disambiguation and a check for possible load-load replay traps.

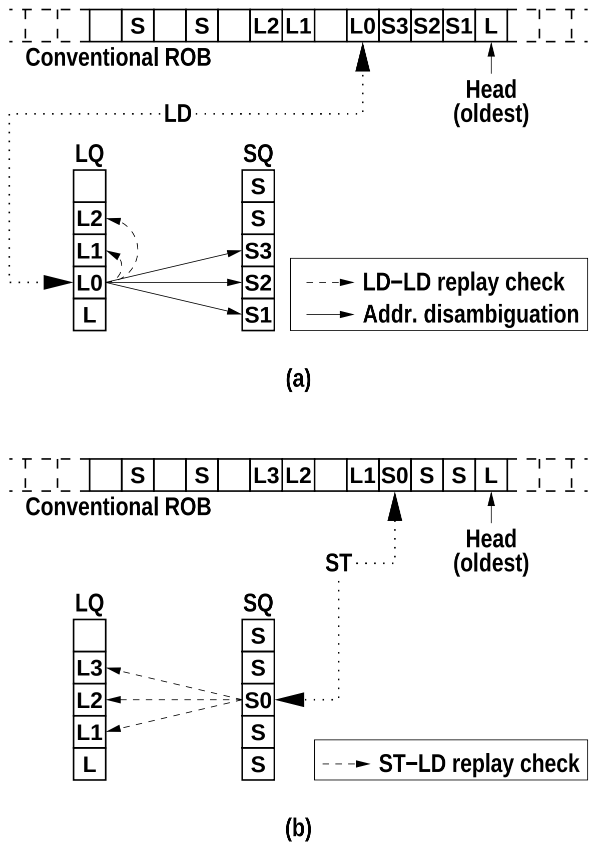

Figure 4: Actions taken on load *(a)* and store *(b)* operations in the conventional load/store unit assumed in this paper. L and Lx stand for Load, while S and Sx stand for Store.

Store-Load Replay

Once the physical address of a store is resolved, it is compared against newer entries in the LQ. The goal is to detect any *exposed* load, namely a newer load whose address overlaps with that of the store, without an intervening store that fully covers the load. Such a load has consumed data prematurely, either from memory or from an earlier store. Thus, the load and all instructions following it are aborted and replayed. This mechanism is called *store-load replay trap* [1].

Figure 4*(b)* shows an example of a check for possible store-load replay traps.

3.1.2 Design with Early Recycling

Following Cherry's philosophy, we want to release LQ and SQ entries as early as it is possible to do so. In this section, we describe the conditions under which this is the case.

Optimized LQ

As explained before, a LQ entry may trigger a replay trap if an older store (or older load in multiprocessors) resolves to an overlapping address. When we release a LQ entry, we lose the ability to compare against its address (Figure 4). Consequently, we can only release a LQ entry when such a comparison is no longer needed because no replay trap can be triggered.

To determine whether or not a LQ entry is needed to trigger a replay trap, we use the U_L and U_S pointers to the ROB (Section 2). Any load that is older than U_S cannot trigger a store-load replay trap, since the physical addresses of all older stores are already known. Furthermore, any load that is older than U_L cannot trigger a load-load replay trap, because the addresses of all older loads are already known.

In a typical uniprocessor environment, only store-load replay traps are relevant. Consequently, as the U_S moves in the ROB, any loads that are older than U_S release their LQ entry. In a multiprocessor or other multiple-master environment, both store-load and load-load replay traps are relevant. Therefore, as the U_S and U_L move in the ROB, any loads that are older than oldest(U_L, U_S) release their LQ entry.[2] To keep the LQ simple, LQ entries are released in order.

Early recycling of LQ entries is not limited by U_B; it is fine for load instructions whose entry has been recycled to be subject to branch mispredictions. Moreover, it is possible to service exceptions and interrupts inside the irreversible set without needing a rollback. This is because LQ entries that are no longer needed to detect possible replay traps can be safely recycled without creating a side effect. In light of an exception or interrupt, the recycling of such a LQ entry does not alter the processor state needed to service that exception or interrupt. Section 3.3 discusses how this blends in a Cherry implementation with recycling of other resources.

Since LQ entries are recycled early, we partition the LQ into two structures. The first one is called *Load Reorder Queue (LRQ)*, and supports the address checking functionality of the conventional LQ. Its entries are assigned in program order at renaming, and recycled according to the algorithm just described. Each entry contains the address of a load.

The second structure is called *Load Data Queue (LDQ)*, and supports the memory access functionality of the conventional LQ. LDQ entries are assigned as load instructions begin execution, and are recycled as soon as the data arrive from memory and are delivered to the appropriate destination register (which the entry points to). Because of their relatively short-lived nature, it is reasonable to assume that the LDQ does not become a bottleneck as we optimize the LRQ. LDQ entries are not assigned in program order, but the LDQ must be addressable by transaction id, so that the entry can be found when the data come back from memory.

Finally, note that, even for a load that has had its LRQ entry recycled, address checking proceeds as usual. Specifically, when its address is finally known, it is compared against older stores for possible data forwarding. This case only happens in uniprocessors, since in multiprocessors the PNR for LQ entries depends on oldest(U_L, U_S).

Optimized SQ

When we release a SQ entry, we must send the store to the memory system. Consequently, we can only release a SQ entry when the old value in the memory system is no longer needed. For the latter to be true, it is sufficient that: (1) no older load is pending address disambiguation, (2) no older load is subject to replay traps, and (3) the store is not subject to squash due to branch misprediction. Condition (1) means that all older loads have already generated their address and, therefore, located their "supplier", whether it is memory or an older store. If it is memory, recall that load requests are sent to memory as soon as their addresses are known. Therefore, if our store is older than U_L, it is guaranteed that all older loads that need to fetch their data from memory have already done so. Condition (2) implies that all older loads are older than oldest(U_S) (typical uniprocessor) or oldest(U_L, U_S) (multiprocessor or other multiple-master system), as discussed above. Finally, condition (3) implies that the store itself is older than U_B. Therefore, all conditions are met if the store itself is older than oldest(U_L, U_S, U_B).[3]

There are two additional implementation issues related to accesses to overlapping addresses. They are relevant when we send a store to the memory system and recycle its SQ entry. First, we would have to compare its address against all older entries in the SQ to ensure that stores to overlapping addresses are sent to the cache in program order. To simplify the hardware, we eliminate the need for such a comparison by simply sending the updates to the memory system (and recycling the SQ entries) in program order. In this case, in-order updates to overlapping addresses are automatically enforced.

Second, note that a store is not sent to the memory system until all previous loads have been resolved (store is older than U_L). One such load may be to an address that overlaps with that of the store. Recall that loads are sent to memory as soon as their addresses are known. The LRQ entry for the load may even be recycled. This case is perfectly safe if the cache system can ensure the ordering of the accesses. This can be implemented in a variety of ways (queue at the MSHR, reject store, etc.), whose detailed implementation is out of the scope of this work.

3.2 Register File

The register file may become a performance bottleneck if the processor runs out of physical registers. In this section, we briefly discuss a conventional design of the register file, and then propose a mechanism for early recycling of registers.

3.2.1 Conventional Design

In our design, we use a register map at the renaming stage of the pipeline and one at the retirement stage. At renaming, instructions pick one available physical register as the destination for their operation and update the renaming map accordingly. Similarly, when the instruction retires, it updates the retirement map to reflect the architectural state immediately after the instruction. Typically, the retirement map is used to support precise exception handling: when an instruction raises an exception, the processor waits until such instruction reaches the ROB head, at which point the processor has a precise image of the architectural state before the exception.

Physical registers holding architectural values are recycled when a retiring instruction updates the retirement map to point away from them. Thus, once a physical register is allocated *at renaming* for an instruction, it remains "pinned" until a subsequent instruction supersedes it *at retirement*. However, a register may become *dead* much earlier: as soon as it is superseded *at renaming*, and all its consumer instructions have read its value. From this moment, and until the superseding instruction retires, the register remains pinned in case the superseding instruction is rolled back for whatever reason, e.g. due to branch misprediction or exception. This effect causes a sub-optimal utilization of the register file.

[2] Note that the LQ entry for the load *equal* to oldest(U_L, U_S) cannot trigger a replay trap and, therefore, can also be released. However, for simplicity, we ignore this case.

[3] Note that it is safe to update the memory system if the store is *equal* to oldest(U_L, U_S, U_B). However, for simplicity, we ignore this case.

3.2.2 Design with Early Recycling

Following Cherry's philosophy, we recycle dead registers as soon as possible, so that they can be reallocated by new instructions. However, we again need to rely on checkpointing to revert to a correct state in case of an exception in an instruction in the irreversible set.

We recycle a register when the following two conditions hold. First, the instruction that produces the register and all those that consume it must be (1) executed and (2) both free of replay traps and not subject to branch mispredictions. The latter implies that they are older than oldest(U_S, U_B) (typical uniprocessor) or oldest(U_L, U_S, U_B) (multiprocessor or other multiple-master system), as discussed above.

The second condition is that the instruction that supersedes the register is not subject to branch mispredictions (older than U_B). Squashing such an instruction due to a branch misprediction would have the undesirable effect of reviving the superseded register. Notice, however, that the instruction can harmlessly be re-executed due to a memory replay trap, and thus ordering constraints around $\{U_L, U_S\}$ are unnecessary. In practice, to simplify the implementation, we also require that the instruction that supersedes the register be older than oldest(U_S, U_B) (typical uniprocessor) or oldest(U_L, U_S, U_B) (multiprocessor or other multiple-master system).

In our implementation, we augment every physical register with a *Superseded* bit and a *Pending* count. This support is similar to [17]. The Superseded bit marks whether the instruction that supersedes the register is older than oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors), which implies that so are all consumers. The Pending count records how many instructions among the consumers and producer of this register are older than oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors) and have not yet completed execution. A physical register can be recycled only when the Superseded bit is set and the Pending count is zero. Finally, we also assume that instructions in the ROB keep, as part of their state, a pointer to the physical register that their execution supersedes. This support exists in the MIPS R10000 processor [23].

As an instruction goes past oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors), the proposed new bits in the register file are acted upon as follows: (1) If the instruction has not finished execution, the Pending count of every source and destination register is incremented; (2) irrespective of whether the instruction has finished execution, the Superseded bit of the superseded register, if any, is set; (3) if the superseded register has both a set Superseded bit and a zero Pending count, the register is added to the free list.

Additionally, every time that an instruction past oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors), finishes executing, it decrements the Pending count of its source and destination registers. If the Pending count of a register reaches zero and its Superseded bit is set, that register is added to the free list.

Overall, in Cherry mode, register recycling occurs before the retirement stage. Note that, upon a collapse step, the processor seamlessly switches from Cherry to non-Cherry register recycling. This is because, at the time the irreversible set is fully collapsed, all early recycled registers in Cherry (and only those) would have also been recycled in non-Cherry mode.

3.3 Putting It All Together

In this section we have applied Cherry's early recycling approach to three different types of resources: LQ entries, SQ entries, and registers. When considered separately, each resource defines its own PNR and irreversible set. Table 1 shows the PNR for each of these three resources.

Resource	PNR Value
LQ entries (uniprocessor)	U_S
LQ entries (multiprocessor)	oldest(U_L, U_S)
SQ entries	oldest(U_L, U_S, U_B)
Registers (uniprocessor)	oldest(U_S, U_B)
Registers (multiprocessor)	oldest(U_L, U_S, U_B)

Table 1: PNR for each of the example resources that are recycled early under Cherry.

When combining early recycling of several resources, we define the dominating PNR as the one which is *farthest* from the ROB head at each point in time. Exceptions on instructions older than that PNR typically require a rollback to the checkpoint; exceptions on instructions newer than that PNR can simply trigger a collapse step so that the processor falls back to non-Cherry mode.

However, it is important to note that our proposal for early recycling of LQ entries is a special case: it guarantees precise handling of extraordinary events even when they occur within the irreversible set (Section 3.1.2). As a result, the PNR for LQ entries need not be taken into account when determining the dominating PNR. Thus, for a Cherry processor with recycling at these three points, the dominating PNR is the newest of the PNRs for SQ entries and for registers. In a collapse step, the dominating PNR is the one that freezes until the ROB head catches up with it.

4 EVALUATION SETUP

Simulated Architecture

We evaluate Cherry using execution-driven simulations with a detailed model of a state-of-the-art processor and its memory subsystem. The baseline processor modeled is an eight-issue dynamic superscalar running at 3.2GHz that has two levels of on-chip caches. The details of the *Baseline* architecture modeled are shown in Table 2. In our simulations, the latency and occupancy of the structures in the processor pipeline, caches, bus, and memory are modeled in detail.

Processor	
Frequency: 3.2GHz	Branch penalty: 7 cycles (minimum)
Fetch/issue/commit width: 8/8/12	Up to 1 taken branch/cycle
I. window/ROB size: 128/384	RAS: 32 entries
Int/FP registers : 192/128	BTB: 4K entries, 4-way assoc.
Ld/St units: 2/2	Branch predictor:
Int/FP/branch units: 7/5/3	Hybrid with speculative update
Ld/St queue entries: 32/32	Bimodal size: 8K entries
MSHRs: 24	Two-level size: 64K entries

Cache	L1	L2	Bus & Memory
Size:	32KB	512KB	FSB frequency: 400MHz
RT:	2 cycles	10 cycles	FSB width: 128bit
Assoc:	4-way	8-way	Memory: 4-channel Rambus
Line size:	64B	128B	DRAM bandwidth: 6.4GB/s
Ports:	4	1	Memory RT: 120ns

Table 2: *Baseline* architecture modeled. In the table, MSHR, RAS, FSB and RT stand for Miss Status Handling Register, Return Address Stack, Front-Side Bus, and Round-Trip time from the processor, respectively. Cycle counts refer to processor cycles.

The processor has separate structures for the ROB, instruction window, and register file. When an instruction is issued, it is placed in both the instruction window and the ROB. Later, when all the input operands are available, the instruction is dispatched to the functional units and is removed from the instruction window.

In our simulations, we break down the execution time based on the reason why, for each issue slot in each cycle, the opportunity to insert a useful instruction into the instruction window is missed (or not). If, for a particular issue slot, an instruction is inserted into the instruction window, and that instruction eventually graduates, that slot is counted as busy. If, instead, an instruction is available but is not inserted in the instruction window because a necessary resource is unavailable, the missed opportunity is attributed to such a resource. Example of such resources are load queue entry, store queue entry, register, or instruction window entry. Finally, instructions from mispredicted paths and other overheads are accounted for separately.

We also simulate four enhanced configurations of the *Baseline* architecture: *Base2*, *Base3*, *Base4*, and *Limit*. Going from *Baseline* to *Base2*, we simply add 32 load queue entries, 32 store queue entries, 32 integer registers, and 32 FP registers. The same occurs as we go from *Base2* to *Base3*, and from *Base3* to *Base4*. *Limit* has an unlimited number of load/store queue entries and integer/FP registers.

Cherry Architecture

We simulate the *Baseline* processor with Cherry support (*Cherry*). We estimate the cost of checkpointing the architectural registers to be 8 cycles. Moreover, we use simulations to derive an average overhead of 52 cycles for a collapse step. Consequently, c_k becomes 60 cycles. If we set the duration of an overhead-free Cherry cycle (T_c) to 5μs, the c_k overhead becomes negligible. Under these conditions, equation 4 yields a total relative overhead (T_o/T_c) of at most one percent, if the separation between exceptions (T_e) is 448μs or more. Note that, in equation 4, we use an average overhead-free Cherry speedup (s) of 1.06. This number is what we obtain for SPECint applications in Section 5. In our evaluation, however, we do not model exceptions. Neglecting them does not introduce significant inaccuracy, given that we simulate applications in steady state, where page faults are infrequent.

Applications

We evaluate Cherry using most of the applications of the SPEC CPU2000 suite [5]. The first column of Table 3 in Section 5.1 lists these. Some applications from the suite are missing; this is due to limitations in our simulation infrastructure. For these applications, it is generally too time-consuming to simulate the reference input set to completion. Consequently, in all applications, we skip the initialization, and then simulate 750 million instructions. If we cannot identify the initialization code, we skip the first 500 million instructions before collecting statistics. The applications are compiled with -O2 using the native SGI MIPSPro compiler.

5 EVALUATION

5.1 Overall Performance

Figures 5 and 6 show the speedups obtained by the Cherry, Base2, Base3, Base4, and Limit configurations over the Baseline system. The figures correspond to the SPECint and SPECfp applications, respectively, that we study. For each application, we show two bars. The leftmost one (*R*) uses the realistic branch prediction scheme of Table 2. The rightmost one (*P*) uses perfect branch prediction for both the advanced and Baseline systems. Note that, even is this case, Cherry uses U_B.

The figures show that Cherry yields speedups across most of the applications. The speedups are more modest in SPECint applications, where Cherry's average performance is between that of Base2 and Base3. For SPECfp applications, the speedups are higher. In this case, the average performance of Cherry is close to that of Base4 and Limit. Overall, with the realistic branch prediction, the average speedup of Cherry on SPECint and SPECfp applications is 1.06 and 1.26, respectively.

If we compare the bars with realistic and perfect branch prediction, we see that some SPECint applications experience significantly higher speedups when branch prediction is perfect. This is the case for both Cherry and enhanced non-Cherry configurations. The reason is that an increase in available resources through early recycling (Cherry) or by simply adding more resources (Base2 to Base4 and Limit) increases performance when these resources are *successfully* re-utilized by instructions that would otherwise wait. Thus, if branch prediction is poor, most of these extra resources are in fact wasted by speculative instructions whose execution is ultimately moot. In perlbmk, for example, the higher speedups attained when all configurations (including Baseline) operate with perfect branch prediction is due to better resource utilization. On the other hand, SPECfp applications are largely insensitive to this effect, since branch prediction is already very successful in the realistic setup.

In general, the gains of Cherry come from recycling resources. To understand the degree of recycling, Table 3 characterizes the irreversible set and other related Cherry parameters. The data corresponds to realistic branch prediction. Specifically, the second column shows the average fraction of ROB entries that are used. The next three columns show the size of the irreversible set, given as a fraction of the used ROB. Recall that the irreversible set is the distance between the PNR and the ROB head (Figure 1). Since the irreversible set depends on the resource being recycled, we give separate numbers for register, LQ entry, and SQ entry recycling. As indicated in Section 3.3, the PNR in uniprocessors is oldest(U_S, U_B) for registers, U_S for LQ entries, and oldest(U_L, U_S, U_B) for SQ entries. Finally, the last column shows the average duration of the collapse step. Recall from Section 3.3 that it involves identifying the newest of the PNR for registers and for SQ entries, and freezing it until the ROB head catches up with it.

	Apps	Used ROB (%)	Irreversible Set (% of Used ROB)			Collapse Step (Cycles)
			Reg	LQ	SQ	
SPECint	bzip2	29.9	24.3	55.8	19.5	292.3
	crafty	28.8	33.4	97.6	28.6	41.9
	gcc	19.1	19.0	82.3	17.8	66.9
	gzip	28.5	65.5	81.7	8.5	47.1
	mcf	30.1	14.6	37.7	13.8	695.6
	parser	30.7	26.1	80.7	21.8	109.2
	perlbmk	12.2	24.6	89.9	20.5	23.3
	vortex	39.3	26.3	87.1	24.9	64.4
	vpr	32.9	25.2	83.6	21.5	165.1
	Average	**27.9**	**28.7**	**77.4**	**19.7**	**167.3**
SPECfp	applu	62.2	61.6	62.4	60.7	411.5
	apsi	76.8	82.3	83.1	81.6	921.1
	art	88.0	54.3	62.6	29.2	1247.3
	equake	41.6	61.6	69.1	57.3	135.2
	mesa	29.8	35.1	44.6	34.6	33.7
	mgrid	65.1	91.5	93.5	91.3	335.9
	swim	59.4	64.8	65.4	64.7	949.1
	wupwise	71.9	90.3	71.2	87.9	190.7
	Average	**61.9**	**67.7**	**78.3**	**63.4**	**528.1**

Table 3: Characterizing the irreversible set and other related Cherry parameters.

Consider the SPECint applications first. The irreversible set for the LQ entries is very large. Its average size is about 77% of the used ROB. This shows that U_S moves far ahead of the ROB head. On the other hand, the irreversible set for the registers is much smaller. Its average size is about 29% of the used ROB. This means that oldest(U_S, U_B) is not far from the ROB head. Consequently, U_B

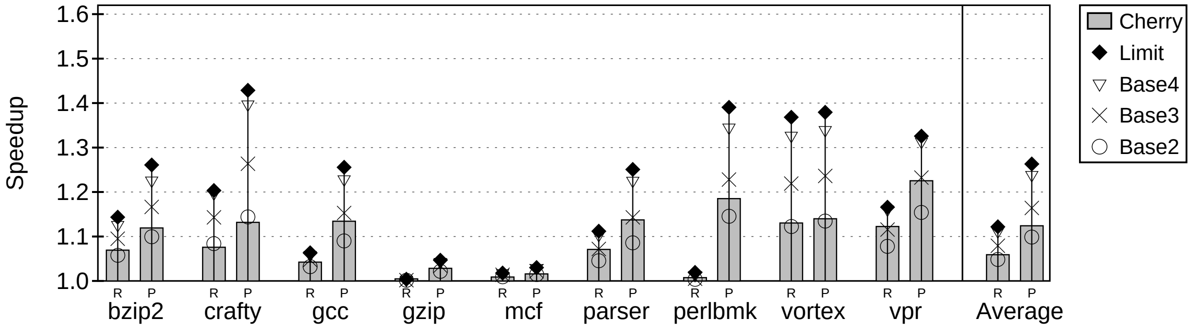

Figure 5: Speedups delivered by the Cherry, Base2, Base3, Base4, and Limit configurations over the Baseline system, for the SPECint applications that we study. For each application, the R and P bars correspond to realistic and perfect branch prediction, respectively.

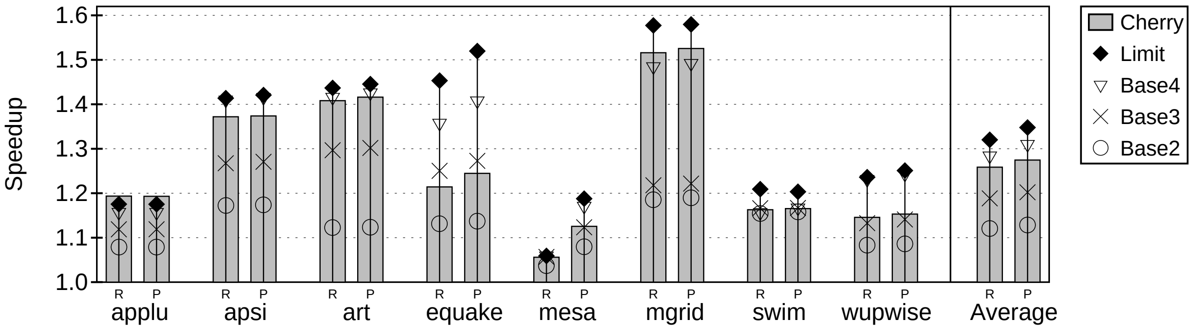

Figure 6: Speedups delivered by the Cherry, Base2, Base3, Base4, and Limit configurations over the Baseline system, for the SPECfp applications that we study. For each application, the R and P bars correspond to realistic and perfect branch prediction, respectively.

is the pointer that keeps the PNR from advancing. In these applications, branch conditions often depend on long-latency instructions and, as a result, they remain unresolved for a while. Finally, the irreversible set for the SQ entries is even smaller. Its average size is about 20%. In this case, PNR is given by oldest(U_L, U_S, U_B) and it shows that U_L further slows down the move of the PNR. In these applications, load addresses often depend on long-latency instructions too.

In contrast, SPECfp applications have fewer conditional branches and they are resolved faster. Furthermore, load addresses follow a more regular pattern and are also resolved earlier. As a consequence, the PNRs for register and SQ entries move far ahead of the ROB head. The result is that Cherry delivers a much higher speedup for SPECfp applications (Figure 6) that for SPECint (Figure 5).

We note that the larger irreversible sets for the SPECfp applications imply a higher cost for the collapse step. Specifically, the average collapse step goes from 167 to 528 cycles as we go from SPECint to SPECfp applications. A long collapse step increases the term c_k in Equation 4, which forces T_c to be longer.

5.2 Contribution of Resources

Figures 7 and 8 show the contribution of different components to the execution time of the SPECint and SPECfp applications, respectively. Each application shows the execution time for three configurations, namely Baseline, Cherry, and Limit. The execution times are normalized to Baseline. The bars are broken down into busy time (*Busy*) and different types of processor stalls due to: lack of physical registers (*Regs*), lack of SQ entries (*SQ*), lack of load queue entries (*LQ*). A final category (*Other*) includes other losses, including those due to branch mispredictions or lack of entries in the instruction window. Section 4 discussed how we obtain these categories.

The Baseline bars show that of the three potential bottlenecks targeted in this paper, the LQ is by far the most serious one. Lack of LQ entries causes a large stall in SPECint and, especially, SPECfp applications.

Our proposal of early recycling of LQ entries is effective in both the SPECint and SPECfp applications. Our optimization reduces most of the LQ stall. It unleashes extra ILP, which in turn puts more pressure on the SQ, register file, and other resources. Even though Cherry does recycle some SQ entries and physical registers, the net effect of our oprimizations is an increased level of saturation on these two resources for both SPECint and SPECfp applications.

One reason why Cherry is not as effective in recycling SQ entries and registers is that their PNRs are constrained by more conditions. Indeed, the PNR for registers is oldest(U_S, U_B), while the one for SQ entries is oldest(U_L, U_S, U_B). In particular, U_B limits the impact of Cherry noticeably.

Overall, to enhance the impact of Cherry, we can improve in two different ways. First, we can design techniques to advance the PNR for SQ entries and registers more aggressively. However, this may increase the risk of a rollback. Second, recycling within the current irreversible set can be done more aggressively. This adds complexity, and may also increase the risk of rollbacks.

5.3 Resource Utilization

To gain a better insight into the performance results of Cherry, we measure the usage of each of the targeted resources. Figure 9 shows cumulative distributions of usage for each of the resources. From top to bottom, the charts refer to LQ entries, SQ entries, integer registers, and floating-point registers. In each chart, the horizontal axis is

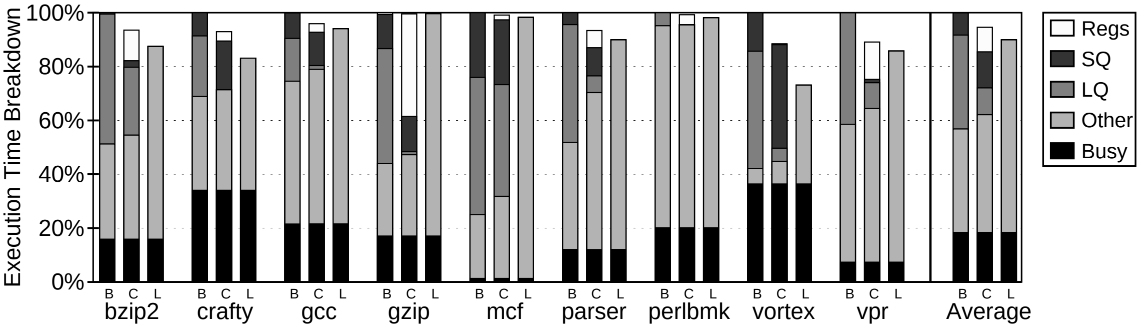

Figure 7: Breakdown of the execution time of the SPECint applications for the Baseline (B), Cherry (C), and Limit (L) configurations.

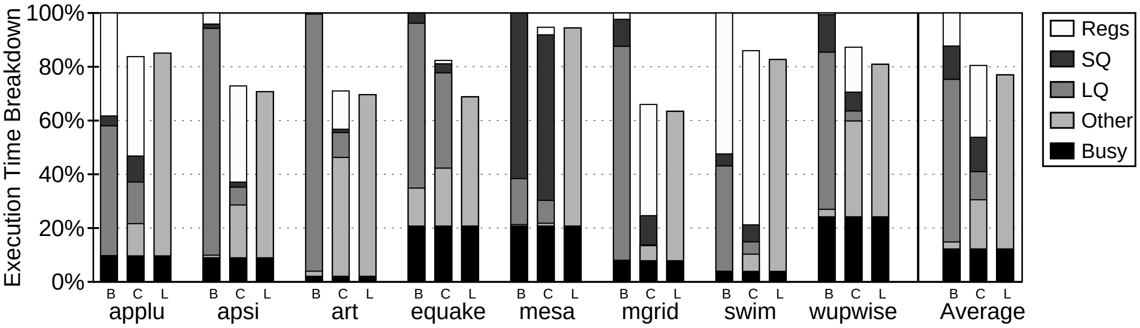

Figure 8: Breakdown of the execution time of the SPECfp applications for the Baseline (B), Cherry (C), and Limit (L) configurations.

the cumulative percentage of time that a resource is allocated below the level shown in the vertical axis. Each chart shows the distribution for the Baseline, Limit, and two Cherry configurations. The latter correspond to the *real* number of allocated entries (*CherryR*) and the *effective* number of allocated entries (*CherryE*). The effective entries include both the entries that are actually occupied and those that would have been occupied had they not been recycled. The difference between CherryE and CherryR shows how effective Cherry is in recycling a given resource. Finally, the area under each curve is proportional to the average usage of a resource.

The top row of Figure 9 shows that LQ entry recycling is very effective. Under Baseline, the LQ is full about 45% and 65% of the time in SPECint and SPECfp applications, respectively. With Cherry, in more than half of the time, all the LQ entries are recycled. We see that the LQ is almost full less than 15% of the time. Moreover, the effective number of LQ entries is significantly larger than the actual size of the LQ.

The second row of Figure 9 shows that, as expected, the recycling of SQ entries is less effective. In SPECint applications, the effective size of the SQ under Cherry surpasses the actual size of that resource significantly in only 6% of the time. However, the potential demand for SQ entries (in the Limit configuration) is much larger. The situation in SPECfp applications is slightly different. The SQ entries are recycled somewhat more effectively.

The last two rows of Figure 9 show the usage of integer (third row) and floating-point (bottom row) registers. In the SPECint applications, the recycling of registers is not very effective. The reason for this is the same as for SQ entries: the PNR is unable to sufficiently advance to permit effective recycling. In contrast, the PNR advances quite effectively in SPECfp applications. The resulting degree of register recycling is very good. Indeed, the effective number of integer registers approaches the potential demand. The potential demand for floating-point registers is larger and is difficult to meet. However, the effective number of floating-point registers in Cherry is larger than the actual size of the register file 50% of the time. In particular, it is more than twice the actual size of the register file 15% of the time.

6 COMBINING CHERRY AND SPECULATIVE MULTITHREADING

6.1 Similarities and Differences

Speculative multithreading (SM) is a technique where several tasks are extracted from a sequential code and executed speculatively in parallel [4, 9, 14, 19, 20]. Value updates by speculative threads are buffered, typically in caches. If a cross-thread dependence violation is detected, updates are discarded and the speculative thread is rolled back to a safe state. The existence of at least one safe thread at all times guarantees forward progress. As safe threads finish execution, they propagate their nonspeculative status to successor threads.

Cherry and SM are complementary techniques: while Cherry uses potentially unsafe resource recycling to enhance instruction overlap *within* a thread, SM uses potentially unsafe parallel execution to enhance instruction overlap *across* threads. Furthermore, Cherry and SM share much of their hardware requirements. Consequently, combining these two schemes becomes an interesting option.

Cherry and SM share two important primitives. The first one is support to checkpoint the processor's architectural state before entering unsafe execution, and to roll back to it if the program state becomes inconsistent. The second primitive consists of support to buffer unsafe memory state in the caches, and either merge it with the memory state when validated, or invalidate it if proven corrupted.

Naturally, both SM and Cherry have additional requirements of their own. SM often tags cached data and accesses with a thread

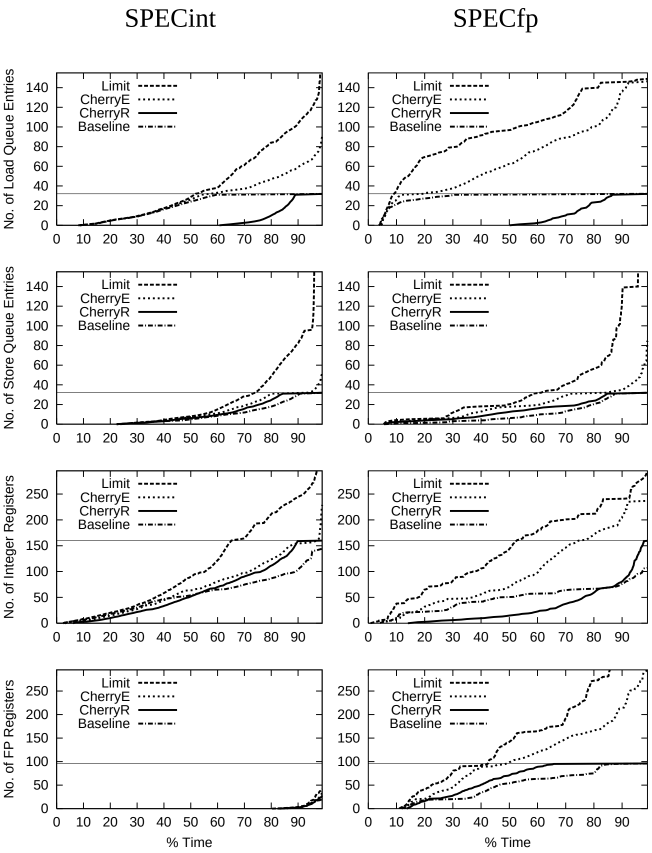

Figure 9: Cumulative distribution of resource usage in SPECint (left) and SPECfp (right) applications. The horizontal axis is the cumulative percentage of time that a resource is used below the level shown in the vertical axis. The resources are, from top to bottom: LQ entries, SQ entries, integer physical registers, and floating-point physical registers.

ID, which identifies the owner or originator thread. Furthermore, SM needs hardware or software to check for cross-thread dependence violations. On the other hand, Cherry needs support to recycle load/store queue entries and registers, and to maintain the PNR pointer.

6.2 Combined Scheme

In a processor that supports both SM and Cherry execution, we propose to exploit both schemes by enabling/disabling speculative execution and Cherry mode in lockstep. Specifically, as a thread becomes speculative, it also enters Cherry mode, and when it successfully completes the speculative section, it also completes the Cherry cycle. Moreover, if speculation is aborted, so is the Cherry cycle, and vice versa. We now show that this approach has the advantage of reusing hardware support.

Enabling and disabling the two schemes in lockstep reuses the checkpoint and the cache support. Indeed, a single checkpoint is required when the thread enters both speculative execution and Cherry mode at once. As for cache support, SM typically tags each cache line with a Read and Write bit which, roughly speaking, are set when the speculative thread reads or writes the line, respectively. On the other hand, Cherry tags cache lines with the Volatile bit, which is set when the thread writes the line. Consequently, the Write and Volatile bits can be combined into one.

With such support, when the thread is speculative, any write sets the Write/Volatile bit. When the thread becomes nonspeculative, all Read bits in the cache are gang-cleared. Then, the processor exits Cherry mode and also gang-clears all Write/Volatile bits.

Special consideration has to be given to cache overflow situations. Under SM alone, the speculative thread stalls when the cache is about to overflow. Under Cherry mode alone, an interrupt informs the processor when the number of Volatile lines in the victim cache exceeds a certain threshold. This advance notice allows the processor to return to non-Cherry mode immediately without overflowing the cache. When we combine both SM and Cherry mode, we stall the processor as soon as the advance notice is received. When the thread later becomes nonspeculative, the thread can resume and immediately return to non-Cherry. Thanks to stalling when the advance notice was received, there is still some room in the cache for the thread to complete the Cherry cycle and not overflow. This strategy is likely to avoid an expensive rollback to the checkpoint.

Another consideration related to the previous one is the treatment of the advance warning interrupt when combining Cherry and SM. Note that the advance notice interrupt in Cherry requires no special handling. Indeed, *any* interrupt triggers the ending of the current Cherry—the advance warning interrupt is special only in that it is signaled when the cache is nearly full. However, when Cherry and SM are combined, the advance warning interrupt has to be recognized as such, so that the stall can be performed before the processor's interrupt handling logic can react to it. This differs from the way other interrupts are handled in SM, where interrupts are typically handled by squashing the speculative thread and responding to the interrupt immediately.

7 RELATED WORK

Our work combines register checkpointing and reorder buffer (ROB) to allow precise exceptions, fast handling of frequent instruction replay events, and recycling of load and store queue entries and registers. Previous related work can be divided into the following four categories.

The first category includes work on precise exception handling. Hwu and Patt [7] use checkpointing to support precise exceptions in out-of-order processors. On an exception, the processor rolls back to the checkpoint, and then executes code *in order* until the excepting instruction is met. Smith and Pleszkun [18] discuss several methods to support precise exceptions. The Reorder Buffer (ROB) and the History Buffer are presented, among other techniques.

The second category includes work related to register recycling. Moudgill et al. [17] discuss performing early register recycling in out-of-order processors that support precise exceptions. However, the implementation of precise exceptions in [17] relies on either checkpoint/rollback for every replay event, or a history buffer that restricts register recycling to only the instruction at the head of that buffer. In contrast, Cherry combines the ROB and checkpointing, allowing register recycling and, at the same time, quick recovery from frequent replay events using the ROB, and precise exception handling using checkpointing. Wallace and Bagherzadeh [22], and later Monreal et al. [16] delay allocation of physical registers to the execution stage. This is complementary to our work, and can be combined with it to achieve even better resource utilization. Lozano and Gao [12], Martin et al. [15], and Lo et al. [11] use the compiler to analyze the code and pass on dead register information to the hardware, in order to deallocate physical registers. The latter approaches require instruction set support: special symbolic registers [12], register kill instructions [11, 15], or cloned versions of opcodes that implicitly kill registers [11]. Our approach does not require changes in the instruction set or compiler support; thus, it works with legacy application binaries.

The third category of related work would include work that recycles load and store queue entries. Many current processors support speculative loads and replay traps [1, 21] and, to the best of our knowledge, this is the first proposal for early recycling of load and store queue entries in such a scenario.

The last category includes work that, instead of recycling resources early to improve utilization, opts to build larger structures for these resources. Lebeck et al. [10] propose a two-level hierarchical instruction window to keep the effective sizes large and yet the primary structure small and fast. The buffering of the state of all the in-flight instructions is achieved through the use of two-level register files similar to [3, 24], and a large load/store queue. Instead, we focus on improving the effective size of resources while keeping their actual sizes small. We believe that these two techniques are complementary, and could have an additive effect.

Finally, we notice that, concurrently to our work, Cristal et al. [2] propose the use of checkpointing to allow early release of unfinished instructions from the ROB and subsequent out-of-order commit of such instructions. They also leverage this checkpointing support to enable early register release. As a result, a large *virtual* ROB that tolerates long-latency operations can be constructed from a small physical ROB. This technique is compatible with Cherry, and both schemes could be combined for greater overall performance.

8 SUMMARY AND CONCLUSIONS

This paper has presented *CHeckpointed Early Resource RecYcling* (*Cherry*), a mode of execution that decouples the recycling of the resources used by an instruction and the retirement of the instruction. Resources are recycled early, resulting in a more efficient utilization. Cherry relies on state checkpointing to service exceptions for instructions whose resources have been recycled. Cherry leverages the ROB to (1) not require in-order execution as a fallback mechanism, (2) allow memory replay traps and branch mispredictions without rolling back to the Cherry checkpoint, and (3) quickly fall back to conventional out-of-order execution without rolling back to the checkpoint or flushing the pipeline. Furthermore, Cherry enables long checkpointing intervals by allowing speculative updates to reside in the local cache hierarchy.

We have presented a Cherry implementation that targets three resources: load queue, store queue, and register files. We use simple rules for recycling these resources. We report average speedups of 1.06 and 1.26 on SPECint and SPECfp applications, respectively, relative to an aggressive conventional architecture. Of the three techniques, our proposal for load queue entry recycling is the most effective one, particularly for integer codes.

Finally, we have described how to combine Cherry and speculative multithreading. These techniques complement each other and can share significant hardware support.

ACKNOWLEDGMENTS

The authors would like to thank Rajit Manohar, Sanjay Patel, and the anonymous reviewers for useful feedback.

REFERENCES

[1] Compaq Computer Corporation. *Alpha 21264/EV67 Microprocessor Hardware Reference Manual*, Shrewsbury, MA, September 2000.

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

[3] J. L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture*, pages 316–325, Vancouver, Canada, June 2000.

[4] L. Hammond, M. Wiley, and K. Olukotun. Data speculation support for a chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, San Jose, CA, October 1998.

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

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[7] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *International Symposium on Computer Architecture*, pages 18–26, Pittsburgh, PA, June 1987.

[8] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization*, Austin, TX, September 2000.

[9] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, September 1999.

[10] 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.

[11] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):922–933, September 1999.

[12] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *International Symposium on Microarchitecture*, pages 293–302, Ann Arbor, MI, November–December 1995.

[13] R. Manohar. Personal communication, August 2002.

[14] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *International Conference on Supercomputing*, pages 365–372, Rhodes, Greece, June 1999.

[15] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *International Symposium on Microarchitecture*, pages 125–135, Research Triangle Park, NC, December 1997.

[16] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Delaying physical register allocation through virtual-physical registers. In *International Symposium on Microarchitecture*, pages 186–192, Haifa, Israel, November 1999.

[17] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *International Symposium on Microarchitecture*, pages 202–213, Austin, TX, December 1993.

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

[19] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[20] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *International Symposium on High-Performance Computer Architecture*, pages 2–13, Las Vegas, NV, January–February 1998.

[21] 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.

[22] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 179–184, Boston, MA, October 1996.

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

[24] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for VLIW processors. In *International Symposium on Microarchitecture*, pages 137–146, Monterey, CA, December 2000.