Rebound: Scalable Checkpointing for Coherent Shared Memory

Rishi Agarwal, Pranav Garg, and Josep Torrellas
University of Illinois at Urbana-Champaign, USA
{agarwa29,garg11,torrella}@illinois.edu
http://iacoma.cs.uiuc.edu

ABSTRACT

As we move to large manycores, the hardware-based *global* checkpointing schemes that have been proposed for small shared-memory machines do not scale. Scalability barriers include global operations, work lost to global rollback, and inefficiencies in imbalanced or I/O-intensive loads. Scalable checkpointing requires tracking inter-thread dependences and building the checkpoint and rollback operations around dynamic groups of communicating processors.

To address this problem, this paper introduces *Rebound*, the first hardware-based scheme for *coordinated local* checkpointing in multiprocessors with directory-based cache coherence. Rebound leverages the transactions of a directory protocol to track inter-thread dependences. In addition, it boosts checkpointing efficiency by: (i) delaying the writeback of data to safe memory at checkpoints, (ii) supporting operation with multiple checkpoints, and (iii) optimizing checkpointing at barrier synchronization. Finally, Rebound introduces distributed algorithms for checkpointing and rollback sets of processors. Simulations of parallel programs with up to 64 threads show that Rebound is scalable and has very low overhead. For 64 processors, its average performance overhead is only 2%, compared to 15% for global checkpointing.

Categories and Subject Descriptors

B [**Hardware**]: B.8 Performance and Reliability,B.8.1 Reliability, Testing, and Fault-Tolerance.

General Terms

Design, Reliability.

Keywords

Scalable Checkpointing, Shared-Memory Multiprocessors, Faults.

1. INTRODUCTION

Most hardware-based machine-checkpointing schemes proposed for coherent shared-memory multiprocessors use *Global* checkpointing, where all processors periodically participate in system-wide checkpoints [13, 17, 18, 19, 23, 27]. In these systems, recovery after a fault entails discarding work from all processors and, in most cases, performing a checkpoint requires a global barrier. Such schemes are well understood and perform acceptably in systems with small processor counts — e.g., up to 16 processors.

As we move toward manycores with many tens of processors [11, 31], however, global checkpointing is not scalable. One reason is the checkpoint-time overhead of synchronizing all the processors and of burstily moving checkpoint data. A second reason is the potentially substantial work wasted to recovery. This is because a fault causes *all* processors to roll back. As we look into the future, chips will use smaller feature sizes and have higher device counts which, combined, will result in lower MTTFs and hence more wasted work. Finally, a third reason is that global checkpointing is inefficient in load-imbalanced loads, as it forces threads that have not done much work to checkpoint. The same is true in I/O-intensive loads, since output I/O is preceded by a checkpoint.

An alternative to global checkpointing is *Coordinated Local* checkpointing [15]. The idea is to coordinate the checkpointing and rollback operations of only the set of processors that have communicated with each other. Coordination among such processors should suffice, given that faults propagate through communication.

With this approach, one needs to track inter-thread data dependencies dynamically, and record the groups of processors that communicate during a certain interval. The processors in such groups will then be checkpointed and rolled back together. This general approach is used in message-passing systems [14]. It is scalable because its overheads depend on the number of processors that communicate with each other, not on the total processor count.

Previous work on hardware-based coordinated local checkpointing for coherent shared memory has focused on snoopy-based machines [1, 2, 3, 12, 33] and, therefore, is not scalable. For example, Banatre *et al.* [2] connected a hardware module to a multiprocessor's bus to detect inter-processor communication.

To provide scalable machine-checkpointing for manycores, this paper contributes with the first hardware-based scheme for coordinated local checkpointing in multiprocessors with directory-based cache coherence. The scheme is called *Rebound.* It has very low performance and power overheads and is scalable.

Rebound introduces several novel features. First, it leverages the transactions of a directory protocol to track inter-thread dependences inexpensively and in a lazy manner. Second, to boost checkpointing efficiency, it introduces architectures to: (i) delay the writeback of data to safe memory at checkpoints, (ii) support operation with multiple checkpoints, and (iii) optimize checkpointing at barrier synchronization. Third, it introduces distributed software algorithms for checkpointing and rollback sets of processors.

We evaluate Rebound with simulations of parallel applications with up to 64 threads. The results show that Rebound is scalable and has very low overhead. In addition, the delayed writebacks at checkpoints and the checkpointing optimization at barrier synchronizations are both very effective, although not additive. During fault-free execution of 64 processors, and without the barrier optimization, Rebound introduces an average performance overhead of only 2% — compared to 15% for global checkpointing.

The paper is organized as follows. Section 2 gives a background; Sections 3 and 4 present Rebound's design; Sections 5 and 6 evalu-

ISCA'11, June 4–8, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0472-6/11/06 ...$10.00.

ate it; Section 7 reviews related work; Section 8 presents a discussion; and Section 9 concludes.

2. CHECKPOINTS & SHARED MEMORY

Checkpointing and rollback [8] is the most popular approach to recovery. Checkpointing can be Global, where all processors periodically cooperate to create a single checkpoint (e.g., [13, 17, 18, 19, 22, 23, 27]), or Local, where there is no global checkpoint (e.g., [1, 2, 3, 9, 28, 29, 33]). In the latter, the system records interactions between processors. If such information is used to force the subset of processors that have communicated with each other to checkpoint together, the system is called Coordinated Local [1, 2, 3, 33]. Otherwise, processors create checkpoints independently, and the system is called Uncoordinated Local [9, 8, 28, 29]. In uncoordinated local checkpointing, rollbacks may risk the domino effect [24], which consists of cascading rollbacks to earlier and earlier points in the program, which processors trigger on each other as they try to find a consistent recovery line.

While conventional designs store the checkpoints in disk, there are a number of proposals that store checkpoints in main memory or memory buffers (e.g., [3, 13, 18, 19, 22, 23, 27]). Such memory can be made safe through data replication, parity, and the use of non-volatile memory. Checkpointing in memory has major performance and power advantages over checkpointing in disk. As the technology for non-volatile memory matures, we expect to see more in-memory checkpointing.

Efficient checkpoints are incremental, meaning that they focus on the data that has changed since the last checkpoint. Such data can be either buffered until the checkpoint and then merged with the memory state [1, 3, 33], or stored in place while copying the safe state into a log until the checkpoint [17, 23, 27, 29], or stored in a different address that is mapped to the original program in lieu of the original address [13, 18, 19, 22].

To provide low-overhead checkpointing for shared memory, Banatre *et al.* proposed the Recoverable Shared Memory (RSM) [2] scheme for in-memory coordinated local checkpointing. In this scheme, a centralized hardware module is attached to the bus of a bus-based multiprocessor. The module snoops every transaction and records inter-thread dependences. When a processor decides to take a checkpoint or roll back, the RSM notifies the processors that have communicated with it in the interval to take a checkpoint or to roll back. A drawback of the scheme is its centralized structure, which inhibits the scalability to systems with many processors, such as manycores that use a directory-based protocol. Moreover, it assumes that faults are detected instantaneously and, therefore, it only needs a single checkpoint at a time.

2.1 How Dependences Affect Checkpointing and Rollback

To construct a consistent recovery line in a shared-memory system with coordinated local checkpointing, we must follow a few rules [2], which are shown in Figure 1. Figure 1(a) shows an inter-thread dependence. First, as Figure 1(b) shows, if the consumer thread checkpoints, then the producer must checkpoint with it. The reason is that the consumer, by checkpointing, certifies the correctness of the work it did up to this point — bar the fault detection latency. By forcing the producer to checkpoint as well, we ensure that the producer will not later find that the data it produced was wrong and need to roll back to before the write. The second rule, shown in Figure 1(c), is that, if the producer rolls back, then the consumer must roll back with it. This is because, since the producer was faulty, the consumer may have consumed wrong data.

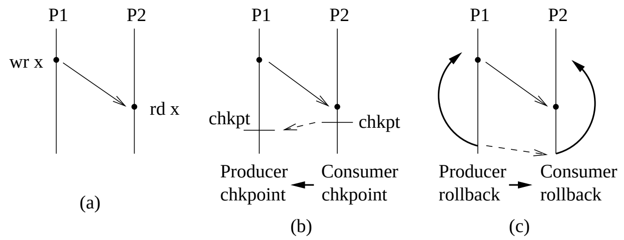


Figure 1: Rules to checkpoint and roll back under coordinated local checkpointing.

3. REBOUND DESIGN

3.1 Main Idea

Rebound provides incremental, in-memory coordinated local checkpointing and rollback in a directory-based cache-coherent manycore. It is scalable and induces very low overhead. Rebound assumes the fault environment described in Section 3.2, where transient and permanent errors can occur anywhere in the chip, and high availability is a requirement. A key idea in Rebound is leveraging directory-based coherence messages to support an efficient, distributed way of recording inter-thread communication. We call the set of processors that communicate with one another in an interval an *Interaction Set*. The processors in an interaction set checkpoint and roll back together, and independently of other sets.

In Rebound, checkpointing and rollback mostly follow the ReVive scheme [23] within each interaction set, rather than globally across all processors. Consequently, creating a checkpoint for an interaction set involves writing back to (off-chip) memory all the dirty lines in the corresponding processors' caches — retaining clean copies in the caches — plus the processors' register state. At each line writeback, the memory controller reads the line's old value from memory and saves it into a software log in memory. In between two checkpoints, any displacement of a dirty cache line to memory also prompts the memory controller to save the old value in the log. Off-chip memory is assumed to be safe, for example through the use of non-volatile memory [25] or DRAM raiding [7].

If a fault is detected and a processor needs to roll back, a software algorithm forces its interaction set to roll back as well. Like in ReVive, rolling back involves invalidating the corresponding processors' caches, copying from the log to memory in reverse order any logged entries from these processors until a safe checkpoint, and restoring the processors' register state at the checkpoint. The fault detection latency determines how far back we need to roll.

The innovations of Rebound are: (i) using directory-protocol transactions to record inter-thread dependences cheaply and lazily; (ii) delaying the writeback of data to safe memory at checkpoints; (iii) supporting multiple checkpoints to handle fault detection latencies; (iv) optimizing checkpointing at barrier synchronization; and (v) developing distributed software algorithms for checkpointing and rolling back interaction sets. In this section and the next, we first describe the fault model and then present each of the contributions.

3.2 Fault Model

To understand the fault model, Figure 2 shows an example of the architecture assumed. The manycore is organized in tiles, where each tile contains a core, private L1-L2 caches, and a directory module [10]. The log is a software structure kept in off-chip memory. The fault model assumes that any part of the chip can suffer transient or permanent faults, even during a checkpointing operation.

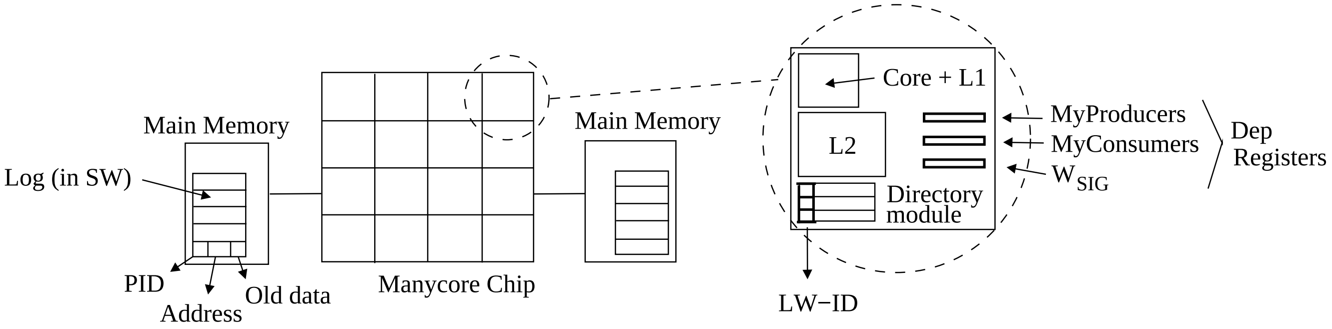


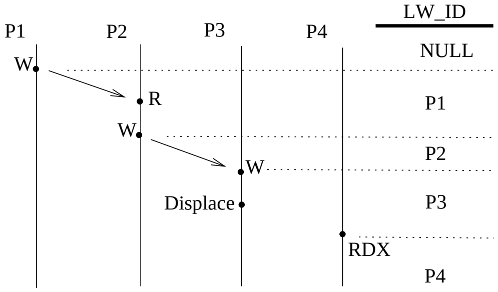
Figure 2: Manycore augmented with the base support for Rebound.

Processor$_i$ action	Previous state of line in cache$_i$	Action	
		LW−ID	MyProducers & MyConsumers

WR line	I, S	LW−ID = PID$_i$	if (LW−ID$_{old}$!= NULL) {
			Set LW−ID$_{old}$ bit in MyProducers$_i$
			Set i bit in MyConsumers$_{LW-ID_{old}}$
			}

RDX line	I		
RD line	I	−−−	

(a)



(b)

Figure 3: Recording inter-thread dependences. All memory accesses in (b) are to the same line.

Data corrupted by faults propagates through communication. For example, a faulty core can write incorrect data to its caches. Faults in caches or interconnect can propagate incorrect data to other cores and caches that read the data, or to off-chip memory when data is written to memory. However, we assume that off-chip memory and logs do not suffer any faults on their own. The techniques needed to ensure that the latter holds, such as ECC, non-volatile memory, or memory raiding, are outside this paper's scope.

There are many techniques to detect faults [20], ranging from general and expensive ones such as core replication and state comparison to more specific and cheap ones such as error detection codes in data paths. We consider the specifics of fault detection to be beyond this paper's scope, except for two aspects. First, since data corrupted by faults propagates through communication, we assume we can detect the scope of the propagation — even in the case of lost or misrouted messages. Consequently, if a core, its cache hierarchy or the data that it wrote to memory have been corrupted, we roll back the core's complete interaction set. Alternately, if a directory module or Rebound metadata has been corrupted, we roll back all the interaction sets. Second, we assume that the fault detection latency has an upper bound of *L* cycles. Consequently, we use the simple model that a checkpoint completed more than *L* cycles ago is safe.

To recover from permanent faults, we additionally assume that there is enough functional hardware remaining to restart the application (e.g., spare cores or network links).

3.3 Basic Operation of Rebound

Figure 2 shows the architecture of a manycore with the base support for Rebound. Although it is not required by Rebound, for this discussion, we assume that the chip has as many directory modules as cores, and that each core has a private L1-L2 cache hierarchy, where L1 is write-through.

The off-chip main memory includes a *software* log for logged data. On chip, the boxes with the thicker lines are the hardware added by Rebound. They include three registers in the L2 cache controller: *MyProducers*, *MyConsumers*, and the Write Signature (W_{SIG}). We call them *Dep* (for dependence) registers. In addition, each entry in the directory module is augmented with a processor ID field called *Last Writer ID* (LW-ID).

3.3.1 Recording Inter-Thread Dependencies

Rebound leverages coherence protocol transactions to track inter-thread dependences off the critical path. To record them, it uses MyProducers and MyConsumers, which have as many bits as processors in the chip. In a processor, bit j of *MyConsumers* is set if, in this checkpoint interval, the processor has produced data that has been consumed by another processor j; Bit j of *MyProducers* is set if, in this checkpoint interval, another processor j has produced data that has been consumed by the local processor. In addition, each directory module entry has an LW-ID field that contains the processor ID of the last writer to the line in this checkpoint interval.

Figure 3(a) shows how we update these structures. Without loss of generality, we assume a MESI protocol. Initially, all structures are null. When a processor first writes a line in this checkpoint interval, the directory not only marks it as owner, but also saves its PID in the LW-ID field of the entry (WR row in Figure 3(a)). Later, when another processor reads the line, the directory forwards the request to the owner processor. There, the L2 controller, as it supplies the line, it sets the bit in its MyConsumers corresponding to the requester. Moreover, as the requester receives the line, it sets the bit in its MyProducers corresponding to the last writer processor (RD row in Figure 3(a)). We have recorded a producer-consumer dependence.

Future readers of the line, as they check the directory, can get the data without communicating with the LW-ID processor. However, the protocol still sends a message to the LW-ID processor, where the L2 updates the corresponding bit in its MyConsumers. Similarly, the reader's L2 updates its MyProducers.

Since coherence protocols work at the cache-line level, Rebound assumes that when a processor writes a line and a second processor reads it, there is a true data dependence. Similarly, when a processor writes a line and a second processor also writes to the line, we have to assume a true data dependence as well, since the second writer can later read silently. Consequently, in the running example, when a new processor writes the line (as in the WR row in Figure 3(a)), the transaction invalidates the current sharers, updates the directory's LW-ID with the writer's ID, and updates MyConsumers and MyProducers as in a read: the old LW-ID processor sets a bit in its MyConsumers, while the new writer sets a bit in its MyProducers.

When a cache line (dirty or otherwise) is displaced from a cache, its LW-ID in the directory is not cleared. Doing so would result in losing the ability to record dependences on the line.

A read transaction may bring a line into a cache in an exclusive state (RDX). In this state, the processor is free to write without informing the directory. Therefore, a RDX transaction, like a WR one, saves the reader's PID in LW-ID (RDX row in Figure 3(a)).

During a checkpoint interval, there may be multiple dependences on a given variable. Figure 3(b) shows an example with different dependences and how LW-ID changes. We will show that the additional coherence traffic needed to maintain MyProducers, MyConsumers, and LW-ID is largely negligible.

Finally, when a processor checkpoints, the L2 controller writes back the dirty lines in L2 to memory, while retaining a clean copy in L2. As the lines are being written back (and the memory controller is logging the old values) the directory clears the Dirty bit but not the LW-ID field. The LW-ID field cannot be cleared because the hardware needs to continue to record dependences with MyProducers and MyConsumers — in case a fault is detected while checkpointing and Rebound has to roll back the local processor and its consumers.

After all the dirty lines are written to memory, the processor's MyProducers and MyConsumers are cleared. We could, at this point, traverse the whole directory and clear the LW-ID field in all the directory entries that have this processor's PID — they are the lines just written back plus other lines written during the checkpoint interval and then displaced or provided to a reader. However, this is too costly. Consequently, we do not do it and allow the LW-ID field to become stale.

3.3.2 Letting Structures Become Temporarily Stale

To improve efficiency, Rebound allows some structures to become temporarily stale. We have seen how LW-ID can become stale. As a result, it is possible that a consumer processor sends a request to the LW-ID processor and the latter concludes (we will see how) that it has not produced the line in this checkpoint interval. In this case, the L2 controller in the LW-ID processor sends a *no-writer* (NO_WR) reply to the directory, which clears LW-ID. The NO_WR reply could be forwarded to the consumer processor, preventing it from updating its MyProducers. However, by that time, MyProducers has already been updated, and reverting it to an accurate state would require costly buffering. Consequently, Rebound lets MyProducers to be a superset of its correct value.

To retain full precision in the presence of potentially stale LW-ID and superset MyProducers, we add a new hardware structure in the L2 controller called *Write Signature* (W_{SIG}). This is a 512–1,024 bit register that encodes, using a bloom filter [4], the addresses of all the lines that the processor has written to (or read exclusively) in the current checkpoint interval. W_{SIG} is cleared at the beginning of every checkpoint interval. If the L2 receives a message that assumes that the local processor is (i) a last writer or (ii) a producer (in the checkpointing protocol of Section 3.3.4) of a line, the controller tests the address for membership in W_{SIG} (using simple logic as in [6]). If the outcome is negative, a NO_WR message is returned. Otherwise, the usual action is taken.

When testing for membership, false negatives are not possible, while false positives are. However, false positives can only result in recording non-existing dependences, possibly causing more rollbacks or checkpoints than strictly needed.

3.3.3 Hardware-Based Logging

The logging algorithm is similar to ReVive [23]. When processors checkpoint, they write back to main memory all of their dirty cached lines. As the memory controller receives each of these lines, it saves the old value of the line in a software log. After all the writebacks, the register state of all the checkpointing processors is also logged. In addition, in between checkpoint times, every time that a dirty line is written to memory (in a cache overflow or when required by the coherence protocol), the memory controller logs the old value of the line. An optimization proposed in ReVive is to log only the first writeback of a line per checkpoint interval [23].

A log entry contains a processor's PID, the old value of the data, and its physical address. Before a set of processors start to checkpoint, one of them stores a stub in the log, to mark where this log starts. Logs can be multi-banked based on address for higher parallelism. In this case, the stub is inserted in all of the banks.

When a set of processors need to roll back, their caches are first invalidated. Then, the logs are read in reverse order, retrieving the entries of only these processors, and writing the values to memory. The operation stops when the corresponding checkpoint-start stub is found. Then, the register state for the processors is restored.

3.3.4 Distributed Checkpointing Protocol

As per Section 2.1, when a processor initiates a checkpoint, it must request that the processors that produced data for it, also checkpoint. Rebound builds the set of producer processors transitively, starting from the initiator. They include the processors in the initiator's MyProducers and, transitively, for each processor there, the set in its own MyProducers. The requests stop propagating when: (i) a processor's MyProducers is null, (ii) a processor is already in the producer set due to cyclic dependences, and (iii) a processor is asked to join the set based on stale information and, therefore, it declines. The last case occurs when processor P1 asks P2 to join, but P2's MyConsumers does not include P1. The reason may be that P1's MyProducers is stale (Section 3.3.2) or that P2 has recently checkpointed and, therefore, cleared its MyConsumers. Overall, the processors collected in this way plus the initiator form the *Interaction Set for Checkpointing* (I_{CHK}).

Rebound identifies the I_{CHK} with a shared-memory *software* algorithm. Figure 4 describes it with an example, where P1 initiates a checkpoint. Figure 4(a) shows the inter-thread dependences and that P4 had checkpointed after providing data. As seen in Figure 4(b), the initiator sends a checkpoint request (*CK?*) to the processors in its MyProducers (P2 and P3), which in turn send it to their MyProducers. Each message from a consumer contains the consumer and the initiator PIDs. The receiver sends an acknowledge (*Ack*) to the consumer, and an *Accept* to the initiator, with the PIDs of its own MyProducers — so that the initiator knows what messages to expect next. A processor receiving a *CK?* request may not be a producer — due to a stale MyProducers or a recent checkpoint. This is the case for P4. In this case, the processor sends a *Decline* message to the initiator (Figure 4(b)). Note that we have described the algorithm in terms of messages for simplicity. In reality, in a shared-memory machine, communication is supported with cross-processor interrupts and memory writes/reads.

A complete checkpoint proceeds as follows. After a processor

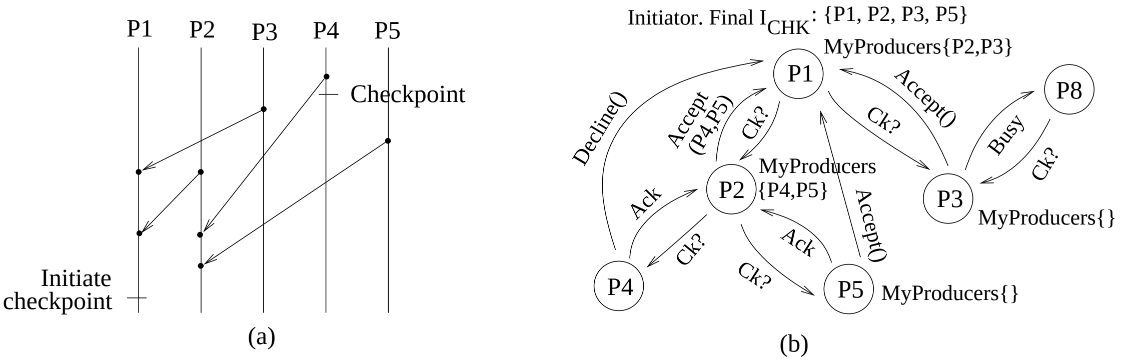


Figure 4: Example of the operation of the distributed checkpointing protocol.

has run a certain number of cycles since its last checkpoint, it initiates the algorithm described to collect its I_{CHK}. Then, it signals all the processors in I_{CHK} to write back their dirty lines. Once all the processors have acknowledged the completion of the writebacks, the initiator signals them to resume normal execution.

Depending on the inter-thread dependence structure, a processor may get *CK?* twice with the same initiator ID. In this case, the processor sends *Ack* and *Accept*, but does not forward *CK?* again. Also, while a processor is participating in a checkpoint, it can receive a *CK?* from another initiator. This is shown in Fig 4(b), where P8 initiates a checkpoint and sends *CK?* to P3 while P3 has already accepted the P1-initiated request. In this case, P3 sends a *Busy* reply to P8. After P3 completes the checkpoint, it clears its MyConsumers. When P8 later retries its request, P3 does not find P8 in its MyConsumers, and sends it a *Decline*. P8 checkpoints alone.

Two checkpoint initiators could intertwine their requests in such a way that each gets some *Accepts* and at least one *Busy*, and none can make progress. To avoid this deadlock, as soon as an initiator receives a *Busy*, it releases all the processors it has already received *Accepts* from. Then, it continues execution for a random number of cycles before attempting a checkpoint again. Finally, another case is when two initiators exchange *CK?* concurrently. A statically-agreed upon priority system causes one initiator to back down.

A fault detected in a processor while checkpointing aborts the whole checkpoint.

3.3.5 Distributed Rollback Protocol

As per Section 2.1, when a processor initiates a rollback, it must request that all the processors that consumed its data also roll back. The set of consumer processors is also built transitively, this time using the initiator's MyConsumers. The total set of consumer processors plus the initiator is the *Interaction Set for Recovery* (I_{REC}).

When a processor initiates a rollback, it follows a software algorithm that is dual to the one in Section 3.3.4. Specifically, the initiator sends a rollback request (*Roll?*) to the processors in its MyConsumers (which in turn send it to their MyConsumers and so on) and waits to receive *Accept*, *Decline*, or *Busy* messages. It will receive a *Decline* if a consumer processor has recently performed an independent rollback and, as a result, cleared its own MyProducers; it will receive a *Busy* if a consumer processor is performing an independent rollback. Once the initiator has collected its I_{REC}, it signals the processors to roll back to their checkpoints. Appendix A shows that the set of the most recent checkpoints of all processors always form a consistent recovery line, and there is no domino effect.

Rolling back a processor involves: (i) clearing its MyProducers, MyConsumers, and W_{SIG}, (ii) invalidating its caches, (iii) restoring to main memory the data from the logs up to the point of the checkpoint, and (iv) restoring the register state at that point. In addition, although not necessary for correctness, as lines are restored to memory, the directories clear those LW-ID fields and Dirty bits that point to the processor. When all the processors acknowledge the completion of rollback to the initiator, the latter signals them to resume normal execution.

If a fault causes the corruption of any of the MyProducers, MyConsumers, W_{SIG}, or LW-ID fields in any processor or directory, Rebound conservatively rolls back *all* the processors in the chip to their checkpoints.

4. BOOSTING EFFICIENCY & USABILITY

We now describe several key Rebound features for efficiency and usability.

4.1 Delayed Writeback of Dirty Cache Lines

A naive design of the checkpointing protocol would require that the participating processors stop the application while the dirty cache lines are written back to memory (Figure 5(a)). This would hurt performance.

To avoid this situation, Rebound performs *Delayed Writebacks*. The idea is that, after all the processors in I_{CHK} agree to checkpoint, they all resume application execution. In the background, the L2 cache controllers write back the dirty lines to memory, avoiding bunching them up. After all the controllers complete their job, the participating processors synchronize again to mark the end of the checkpoint. This must occur before the system wants to start a new checkpoint (Figure 5(b)).

With this approach, since we overlap the writebacks with useful work from the next checkpoint interval, the performance is higher. However, since the writebacks proceed more slowly, the checkpoint duration is longer. If, before the end of the writebacks, a fault is detected or a rollback request is received, both the current checkpoint interval and the previous one need to be rolled back.

To implement this technique, each L2 controller needs two sets of Dep registers (MyProducers, MyConsumers, and W_{SIG}) — a primary and a secondary set. Moreover, each L2 cache line has a Delayed Writeback bit (*Delayed* for short). As soon as the processor has initially synchronized with the other checkpointing processors, the hardware sets the Delayed bit for all the dirty lines in L2, and the application is invoked again. As the application resumes, the coherence protocol switches to using the secondary set of Dep registers. In the background, the L2 cache controller writes back the lines with the Delayed bit set, clearing the bit in turn. When the controller finishes its task, the processor is interrupted. Then, all the checkpointing processors synchronize again. The hardware then clears the primary set of Dep registers, leaving them ready for the next checkpoint interval.

Several events may occur while the delayed writebacks are in

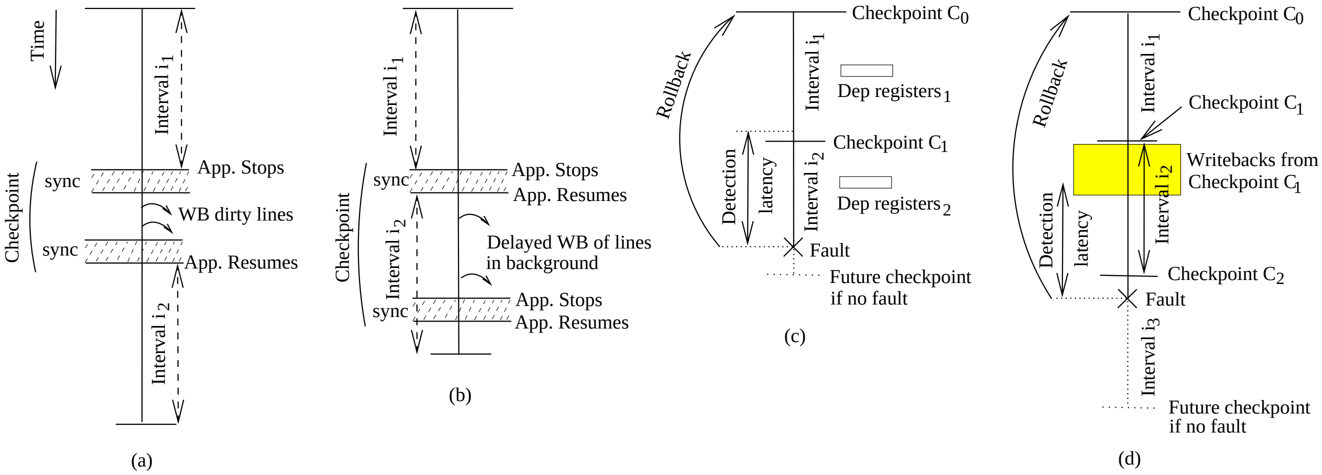


Figure 5: Support for delayed writebacks (a and b) and multiple checkpoints (c and d).

progress. First, the processor may write to a line that is still marked Delayed. In this case, the line is immediately written back and the Delayed bit cleared before the write can complete. Second, the L2 controller may receive a "are you the last writer?" request from a consumer. In this case, the hardware checks the two W_{SIG} to see which interval produced the data. For the one that matches, it sets MyConsumers. If the requested address is in both signatures, the hardware updates MyConsumers for the later checkpoint interval, which is conservative if a rollback is required later. Finally, the processor may receive an external request to checkpoint. In this case, it responds with a *Nack* and the controller speeds-up the writeback of the Delayed lines. The hardware needs to complete the delayed checkpoint before it can accept any checkpointing request. A *Nack* prompts the requester to retry.

In cache-hierarchy buffers, delayed writebacks have lower priority than and are bypassed by the normal reads and writes. Moreover, we envision hardware in the L2 controller that measures the round trip latencies of cache misses. If latencies are high, the cause may be the frequent writebacks. Consequently, the controller can slow-down the writeback frequency. If latencies are low, the opposite can be done. A simpler, coarser approach to detect writeback-induced slowdowns is to monitor changes in program IPC.

4.2 Multiple Checkpoints

In any realistic environment, the fault detection latency is not zero and, therefore, we need to keep multiple checkpoints. In Rebound, this means keeping multiple sets of Dep registers.

Given a fault-detection latency *L*, we set the checkpoint interval to be larger than *L*. Consequently, in a fault, we theoretically need to roll back at most two intervals and, therefore, only need two sets of Dep registers. This is shown in Fig 5(c), where a fault rolls back the execution of both checkpoint intervals i_2 and i_1. Note that, in the example, we have written back to memory the data generated in interval i_1 before we could guarantee that i_1 will not need to be rolled back. This is fine because i_1's updates can be undone thanks to the log. However, the Dep registers for i_1 cannot be recycled before we can guarantee that i_1 will not need to be rolled back. To see why, assume that another thread reads data generated in i_1. In this case, MyConsumers from i_1 needs to record it. This is necessary because if i_1 is rolled back, we must roll back all of the consumers of i_1's data as well.

In reality, we need more sets of Dep registers. A reason is that the processor may be asked by other processors to checkpoint multiple times. For every new checkpoint interval i_n, a new set of Dep registers is required. Dep registers for i_n can only be recycled when the checkpoint that follows i_n completed at least *L* cycles ago.

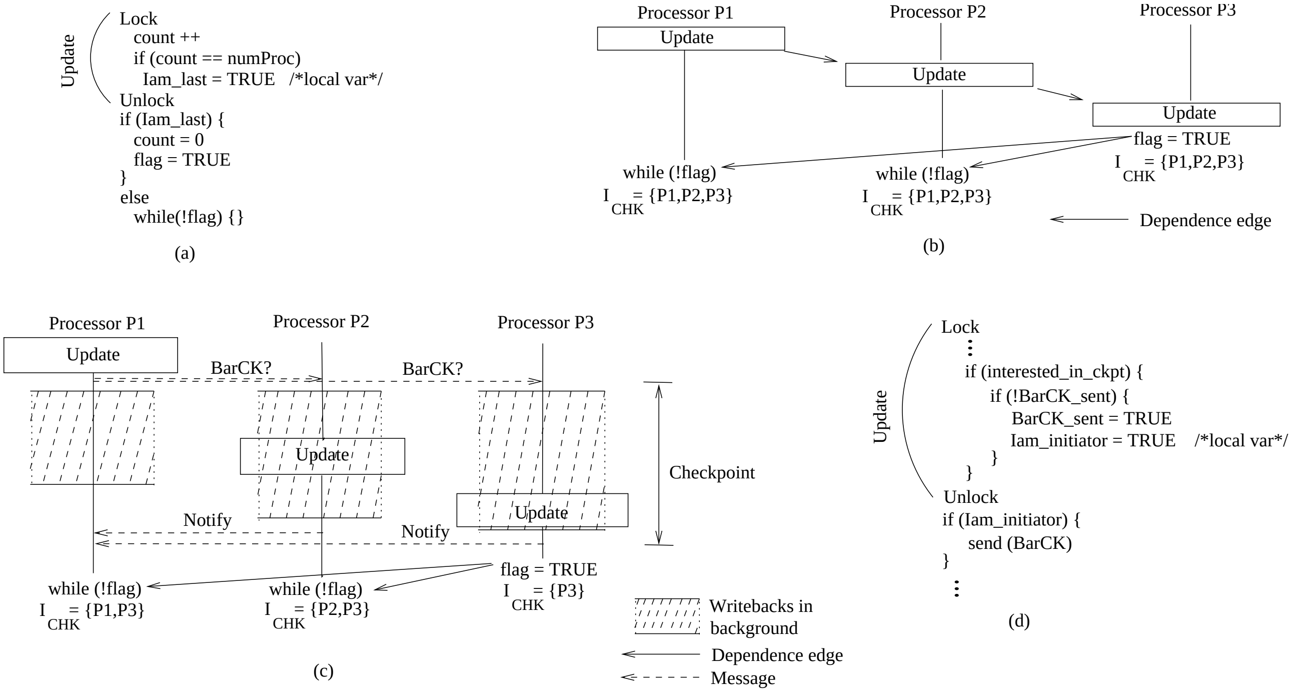
Further, as argued before, the use of delayed writebacks requires the allocation of one additional set of Dep registers. This can be seen in Figure 5(d). When the fault is detected, we subtract *L* cycles and find that interval i_2 may be polluted. In addition, since at that time, the L2 controller was still writing back data from interval i_1, i_1 may also be polluted. Consequently, we need to roll back three intervals (i_3, i_2, and i_1). In general, to compute the number of intervals to roll back, we subtract *L* from the current time to estimate when the fault occurred. We then roll back all the intervals up to (and including) the one executing at that time, plus one more — in case data from the previous interval was still being written back in the background when the fault occurred.

Each processor keeps several sets of Dep registers and keeps recycling them. When a processor wants to initiate a new checkpoint interval and is out of Dep registers, it stalls. It waits until the following is true for the interval i_n that owns the earliest set of Dep registers: the checkpoint that follows i_n completed at least *L* cycles ago — including the writebacks. At that point, it can recycle the Dep registers.

To fully understand the operation of Rebound under multiple checkpoints, consider four events that can occur. The first one is when the L2 controller receives a "are you the last writer?" protocol message. The L2 controller checks the address for membership in all the W_{SIG} in use, in reverse age, starting from the latest W_{SIG}. As soon as one matches, the controller sets the bit in MyConsumers for that checkpoint interval and stops. As indicated before, this is a conservative approach. If none matches, the directory is informed to clear LW-ID.

A second event is when the processor detects a fault. The processor rolls back to the latest checkpoint that fully completed at least *L* cycles ago (Figure 5(d)) — including delayed writebacks. Then, it reads the MyConsumers registers of all the checkpoint intervals that it is rolling back, performs their logical OR to collect all the consumer processors, and sends rollback requests to all of them.

A third event is when the processor receives a rollback request from one of its producers. One approach would be to use MyProducers to check which local checkpoint interval consumed data from the requester and only unroll that interval (and later ones). However, this approach is too complex: it may result in the processor receiving multiple, successive rollback requests, as the complete Interaction Set for Recovery (I_{REC}) for this operation is being formed. Instead, it is simpler for the local processor to roll back



```
Lock
  count ++
  if (count == numProc)
    Iam_last = TRUE   /*local var*/
Unlock
if (Iam_last) {
  count = 0
  flag = TRUE
}
else
  while(!flag) {}
```

(a)

(b)

(c)

```
Lock
  ⋮
    if (interested_in_ckpt) {
        if (!BarCK_sent) {
            BarCK_sent = TRUE
            Iam_initiator = TRUE    /*local var*/
        }
    }
Unlock
if (Iam_initiator) {
    send (BarCK)
}
  ⋮
```

(d)

Figure 6: Optimization of the checkpoint at barriers.

to the latest checkpoint that fully completed at least *L* cycles ago (including delayed writebacks). Then, it sends rollback requests to all of its consumers, which will do the same. Appendix A shows that this algorithm produces a consistent recovery line.

Finally, the processor may receive a checkpoint request from a consumer. The processor checks if this consumer has indeed consumed data from the *latest* local checkpoint interval (by checking MyConsumers). If it has, the local processor agrees to participate and sends a checkpoint request to its own producers in this checkpoint interval (by checking MyProducers). Otherwise, it sends a *Decline*. Previous local checkpoint intervals do not need to be considered because they have already been checkpointed.

4.3 Optimizing Checkpointing at Barriers

Figure 6(a) shows a simple implementation of a barrier. It is composed of the *Update* critical section that increments the count of processors that have arrived, and then a spin on a flag until the last arriving processor writes to it. This pattern creates a dependence chain that includes all the processors. As shown in Figure 6(b), if any processor initiates a checkpoint after the barrier, it finds that all the processors that synchronized are in its Interaction Set for Checkpointing (I_{CHK}). Consequently, global barriers induce global checkpoints.

To reduce the overhead of these checkpoints and largely hide them behind the barrier imbalance time, Rebound introduces the *Barrier* optimization. This optimization is especially effective when the system does not support the Delayed Writebacks optimization of Section 4.1 and, therefore, the whole overhead of the checkpoint would otherwise appear in the critical path. However, it is still effective even with the Delayed Writebacks optimization.

The optimization consists of triggering a proactive checkpoint at the barrier. Specifically, when a processor reaches a barrier and completes the Update section, it initiates a global checkpoint by sending a special *BarCK* signal to all the other processors (Figure 6(c)). After all processors have responded with an accept message, the initiator tells them to begin writing back to memory the dirty lines from their caches in the background — as they continue to execute the program. In this way, the latency of the checkpoint's writebacks is hidden, either behind spins at the flag of the barrier (for the processors with little work like P1) or behind the execution of code that brings the processor to the barrier (for the processors with more work, like P2 and P3). This is shown in Figure 6(c).

After a processor has both executed the Update section of the barrier and completed the writebacks in the background, it notifies the initiator processor. Note that the processor that arrives at the barrier last is not allowed to set the flag yet, while the other processors are free to spin on the flag. When the initiator has received all the notifications, it signals all of the processors that the checkpoint is completed and that they can continue. At this point, the last arriving processor will write the flag and release all the processors. With this scheme, processors leave the barrier with a very small I_{CHK}. Specifically, as shown in Figure 6(c), the I_{CHK} of a processor includes itself and the processor that set the flag.

For this algorithm to work, we must ensure that only one processor acts as checkpoint initiator. In addition, it is possible that some processors have recently checkpointed and, therefore, are not interested in checkpointing. These processors will decline to initiate the checkpoint, but will participate if another processor wants to checkpoint. Consequently, we modify the barrier by adding the code shown in Figure 6(d). *BarCK_sent* is a global variable that indicates whether a processor has already sent the *BarCK* messages. Inside the Update section, if a processor wants to initiate a checkpoint, it checks *BarCK_sent* and, if it is clear, it sets it. In this case, after exiting the critical section, the processor sends the *BarCK* messages. By the time *BarCK* messages are sent, some processors that were not interested in checkpointing may be already spinning in the flag. They are also forced to participate in the checkpoint.

5. EVALUATION SETUP

To evaluate Rebound, we built an analysis tool using Pin [16]. The output of Pin is connected to a detailed multi-processor architecture simulator based on SESC [26] that is interfaced to the

Architecture Parameters
System: Manycore with up to 64 cores
Core: single−issue at 1GHz
L1: 16KB, 4−way assoc, 32B line Private, write−through, 16−entry MSHR Hit delay: 2 cycles round trip
L2: 256KB, 8−way assoc, 32B line Private, write−back, 16−entry MSHR Hit delay: 8 cycles round trip Miss delay: To other L2s: 60 cycles round trip (avg) To main memory: 200 cycles round trip
Multistage interconnect
Directory module: full map
Memory: 2 channels DDR2 DRAMs. Datarate 667 MHz
W_SIG: 1024 bits, similar to Notary's PBX
Chip area: 200 sq. mm. at 45nm
Hierarchical clock tree

Configurations Evaluated
Rebound: Proposed scheme without Barrier optimization.
Rebound_NoDWB: Rebound without the Delayed Writebacks.
Rebound_Barr: Rebound with the Barrier optimization.
Rebound_NoDWB_Barr: Rebound without the Delayed Writebacks and with the Barrier optimization.
Global: Global checkpointing (baseline).
Global_DWB: Global with the Delayed Writebacks.

Checkpointing Parameters
Checkpoint interval: 4M instructions (About 5−−8 ms)
Number of Dep register sets: 4 maximum
Availability target: > 99.999% available
Maximum recovery latency: approx. 860 ms

(a)

Application	Problem Size
SPLASH−2:	
LU−C, LU−NC	1500*1500 matrix 16*16 blks
Volrend	head
Water−Sp	1000 molecules
Water−Nsq	1000 molecules
Raytrace	car
FFT	1M points
FMM	16K particles
Radix	2048K ints, radix 1024
Barnes	32K particles
Cholesky	tk29.O
Radiosity	room
Ocean	258 * 258 ocean
PARSEC:	
Blackscholes	simlarge
Fluidanimate	simlarge
Ferret	simlarge
Streamcluster	simlarge
Apache	ab tool

(b)

Figure 7: Simulated system configuration (a) and applications evaluated (b).

DRAMsim [32] main memory simulator. We model a manycore with up to 64 cores like the one in Figure 2. The cores issue and commit one instruction per cycle. They overlap memory accesses with instruction execution through the use of a reorder buffer. The architectural parameters of the simulated machine are shown in Figure 7(a). The simulator has integrated models of power from CACTI [30] and Wattch [5] that have been updated with data from ITRS 2010 to model static and dynamic power at 45nm.

We evaluate Rebound on SPLASH-2, some applications from PARSEC, and Apache. The applications and problem sizes are listed in Figure 7(b). To simulate more threads (up to 64) than the number of processors in the largest machine that we have available (24), we interface our Pin tool to a customized Pthread scheduling library. This library maintains instruction queues that are then scheduled in parallel on the available processors. Consequently, we evaluate SPLASH-2 for up to 64 threads. However, as this library does not work with PARSEC and Apache, we can only evaluate these two workloads for up to 24 threads.

We target a highly available system. Following ReVive [23], our goal is an availability greater than 99.999%. This means that, if there is one error per day, the recovery latency must be no higher than 860 ms. Although ReVive is a global checkpointing scheme, it uses a generally similar in-memory checkpointing approach as Rebound. ReVive found that the recovery latency is largely determined by the restoration of the logged data. ReVive attained the 860 ms recovery latency with a 100 ms checkpoint interval for 16 processors. Since we evaluate Rebound for 64 processors, to attain a maximum recovery latency that is no higher, we need a checkpoint interval that is about one order of magniture shorter. Consequently, we set the checkpoint interval to 4 million instructions, which corresponds to a 5–8 ms checkpoint interval.

We evaluate the configurations shown in Figure 7(a). *Rebound* is our proposed local checkpointing scheme of Sections 3 and 4, *without the Barrier optimization* of Section 4.3. *Rebound_NoDWB* is Rebound without the Delayed Writebacks optimization of Section 4.1. *Rebound_Barr* is Rebound with the Barrier optimization. *Rebound_NoDWB_Barr* is Rebound without the Delayed Writebacks and with the Barrier optimization.

We compare Rebound to *Global*, a global checkpointing scheme that we use as baseline. *Global* uses the same manycore architecture as Rebound, namely that of Figure 2. At periodic intervals equal to the checkpoint interval, an interrupt is sent to all processors, which then synchronize. Then, they all write back their dirty cache lines and their register state. Finally, they synchronize again and resume execution.

We also evaluate *Global_DWB*, which is *Global* with the Delayed Writebacks optimization.

6. EVALUATION

We evaluate the size of Rebound's interaction set for checkpointing, its checkpointing overhead during error-free execution, its scalability, the effect of I/O, its power consumption, and some other characteristics.

6.1 Size of Interaction Set for Checkpointing

The size of the Interaction Set for Checkpointing (I_{CHK}) is the number of processors that checkpoint together. Figures 8 and 9 show the average I_{CHK} size for PARSEC/Apache and SPLASH-2, respectively, as a percentage of the total number of processors running. For PARSEC/Apache, the data is for 24-processor runs, while for SPLASH-2, Figure 9 shows data for 32 and 64-processor runs. The figures show data for *Global* and *Rebound*.

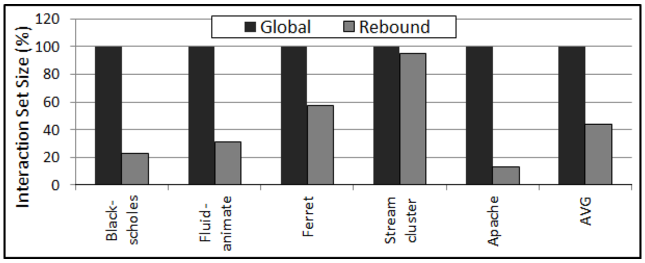
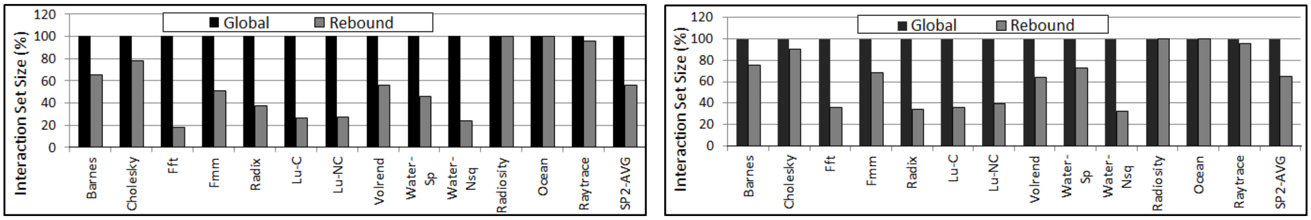


Figure 8: Average size of the Interaction Set for Checkpointing for PARSEC and Apache for 24-processor runs.

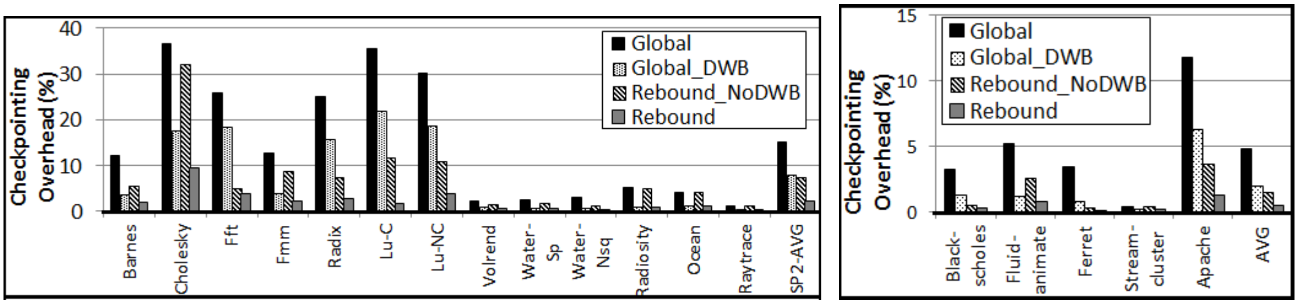
For *Global*, the I_{CHK} size is always 100%. For *Rebound*, the average I_{CHK} size is a characteristic of the application. In codes that have communication locality such as Blackscholes and Apache, it is about 20%; in codes that have a large number of dynamic locks or very frequent barriers such as Ocean and Raytrace, it is about 100%. For example, Ocean has a barrier every 50k instructions. On average for our codes, Rebound reduces the I_{CHK} size to about



(a)

(b)

Figure 9: Average size of the Interaction Set for Checkpointing for SPLASH-2 for 32-processor runs (a) and 64-processor runs (b).



(a)

(b)

Figure 10: Checkpointing overhead (as a fraction of the execution time) during error-free execution in 64-processor SPLASH-2 runs (a) and 24-processor PARSEC and Apache runs.

40% for PARSEC and Apache, and to about 60% for SPLASH-2. Going from 32 to 64 processors increases the I_{CHK} size only a little. Overall, we consider these results to be positive: even for application suites that were not specifically written for clustered communication, the I_{CHK} size decreases by half. Scalable applications for larger machines are very likely to show much more communication clustering.

6.2 Overhead During Error-Free Execution

Figure 10 shows the checkpointing overhead (as a fraction of the execution time) during error-free execution in SPLASH-2 and PARSEC/Apache. The figure shows bars for *Global*, *Global_DWB*, *Rebound_NoDWB*, and *Rebound*. The overhead is a function of the number of processors checkpointing together and the number of dirty lines written back to the memory at a checkpoint.

We can see that there is variation across applications. However, on average, *Global* has substantial overhead, while *Rebound* practically eliminates it all. Specifically, for SPLASH-2, *Global*'s overhead is 15%, while *Rebound*'s is only 2%. For PARSEC/Apache, *Global*'s overhead is 5%, while *Rebound*'s is 0.5%. The figure also shows that delayed writebacks account for about one third of the impact of *Rebound* in SPLASH-2. Indeed, the average overhead of *Rebound_NoDWB* in SPLASH-2 is 7%. Therefore, we suggest supporting delayed writebacks in local checkpointing. However, simply adding delayed writebacks to *Global* is not good enough. For example, the average overhead of *Global_DWB* in SPLASH-2 is 8%. We need both local checkpointing and delayed writebacks. If any of the two features is not supported, not only does the average overhead increase, but the overhead of some applications becomes intolerably high as well.

Barrier Optimization. Figure 11 takes all the barrier-intensive applications and shows the impact of the Barrier optimization on the checkpointing overhead. From left to right, the figure shows bars for *Global*, *Rebound_NoDWB*, *Rebound_NoDWB_Barr*, *Rebound*, and *Rebound_Barr*. The difference between the second and the third bars is the impact of the Barrier optimization; the difference between the second and the fourth bars is the impact of the delayed writebacks. Looking at the average, we see that both features have approximately similar impacts, although delayed writebacks is a bit better. Combining both features (fifth bars) does not add-up their individual impacts. Given the lower applicability of the Barrier optimization, we choose to include delayed writebacks in our *Rebound* proposal and not the Barrier optimization.

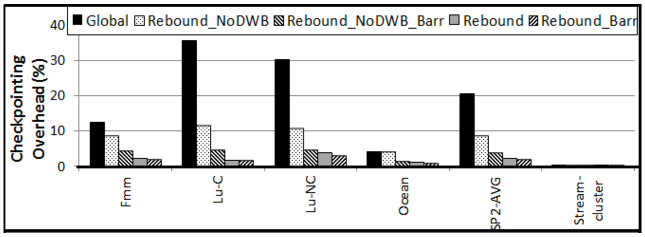


Figure 11: Impact of the Barrier optimization on the checkpointing overhead.

Overhead Breakdown. To understand the checkpointing overhead, Figure 12 breaks it down into four categories. *WBDelay* is the stall time when a processor writes back its dirty lines at a checkpoint. *WBImbalanceDelay* is the stall time when a processor waits for the other checkpointing processors to complete their writebacks *after* it has already finished its own writebacks. *SyncDelay* is the synchronization cost to coordinate the checkpointing processors. Finally, *IPCDelay* is the processor slowdown (intuitively, the "IPC decrease") caused by background traffic induced by delayed writebacks or other processor's checkpoints. The figure shows bars for *Global*, *Rebound_NoDWB*, and *Rebound*, all normalized to *Global*.

We see that, in *Global* and *Rebound_NoDWB*, *WBDelay* and *WBImbalanceDelay* dominate. On the other hand, in *Rebound*, since the writebacks are issued in the background, *IPCDelay* is the main contributor to the overhead. *SyncDelay* is minor.

6.3 Scalability Analysis

To assess scalability, we measure the changes in checkpointing overhead, energy consumed due to checkpointing, and recovery latency as we increase the number of processors. We compare *Global*, *Rebound_NoDWB*, and *Rebound* running the SPLASH-2

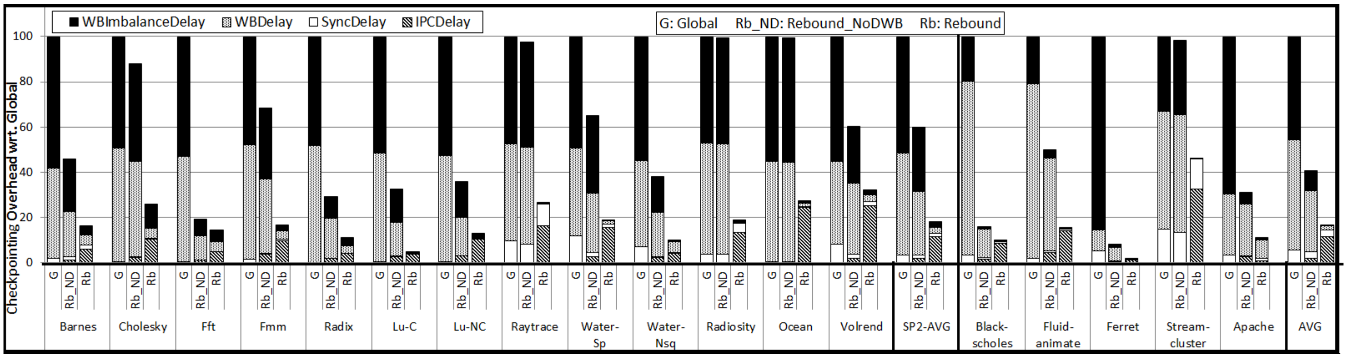
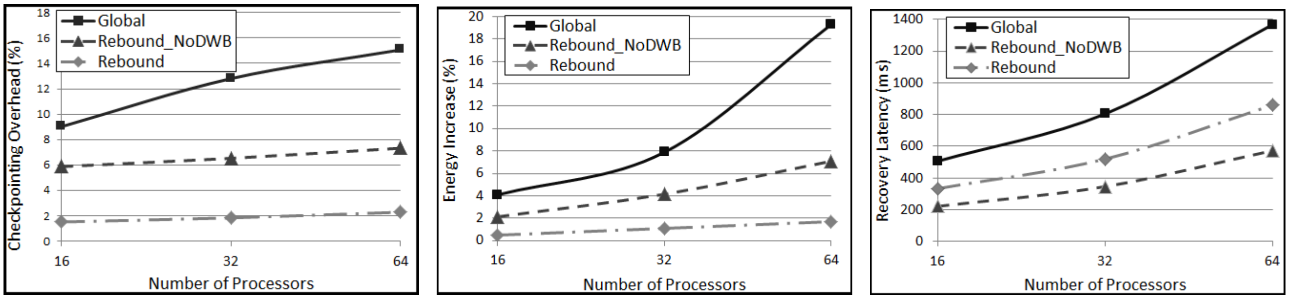


Figure 12: Breakdown of the checkpointing overhead. The SPLASH-2 codes run on 64 processors, while the other codes run on 24 processors. The bars are normalized to *Global*.



(a) (b) (c)

Figure 13: Changes in checkpointing overhead (a), energy consumption increase due to checkpointing (b), and fault recovery latency (c) for SPLASH-2 as we increase the processor count.

codes for 16, 32, and 64 processors. Figure 13(a) shows the checkpointing overhead. We can see that the local schemes scale much better than *Global* because they operate on subsets of processors. The very mild slope of the *Rebound* curve confirms that this scheme scales to large processor counts.

Figure 13(b) shows the increase in on-chip energy consumption (both dynamic and static) due to checkpointing. This includes the effect of both the additional hardware structures and the messages. The local schemes are more efficient than *Global* because they transfer less data. Importantly, they are more scalable. *Rebound* is more efficient and scalable than *Rebound_NoDWB* because it overlaps the writebacks with useful work. Overall, at 64 processors, *Rebound* checkpointing increases the energy consumed by the chip by 2%, while *Global* checkpointing increases it by 19%.

Figure 13(c) shows the average recovery latency on a transient fault right before starting a checkpoint. We see that the local schemes take less time than *Global* because they restore less data. They are also more scalable. *Rebound* takes longer than *Rebound_NoDWB* because, by performing delayed writebacks, a rollback requires the undo of one additional checkpoint. Overall, at 64 processors, the recovery latency of *Rebound* is well under one second, delivering about 99.999% availability for one of these faults per day.

6.4 Estimated Impact of Output I/O

Since an output I/O is preceded by a checkpoint, I/O-intensive codes hurt *Global*: many processors must checkpoint without having done much work. Schemes that checkpoint smaller sets of processors like *Rebound* are less affected. To estimate this effect, we take 5 codes that have a relatively low I_{CHK} size, set the checkpoint interval to 5M cycles, and force one processor of the 64 to initiate a checkpoint (as if it was performing output I/O) every 2.5M cycles. Figure 14 shows the resulting average checkpoint interval. The global scheme with I/O (*Global-I/O*) reduces its average checkpoint interval to 2.5M cycles, while the local one with I/O (*Rebound-I/O*) keeps the average above 4M cycles. The latter runs more efficiently. We can see that *Rebound* is much less disrupted by a frequently-checkpointing thread.

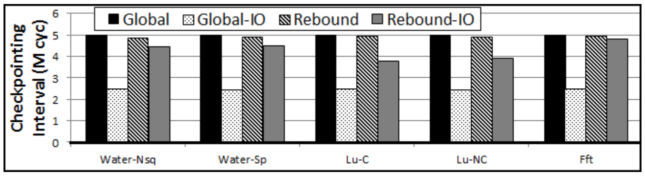


Figure 14: Effect of output I/O on the checkpoint interval.

6.5 Power Analysis

Figure 15 shows the estimated on-chip power consumption (both dynamic and static) in the *Global*, *Rebound_NoDWB*, and *Rebound* systems. The data corresponds to the average of the SPLASH-2 applications running with 64 processors. We see that *Rebound_NoDWB* and *Rebound* consume 2% and 4%, respectively, more power than *Global*. These numbers include a 1.3% power cost to maintain the additional hardware structures. The rest of the increase is due to the more efficient execution (e.g., writebacks in the background in *Rebound*). It can be shown that *Rebound* reduces the ED^2 (energy delay square product) of *Global* by 27%.

6.6 Miscellaneous Characterization

Table 1 characterizes three properties of Rebound for each of the 18 applications. When a processor is asked if it wrote a line in this checkpoint interval, it checks its W_{SIG}. If the line is not present but, due to aliasing, a match is detected, we may be unnecessarily increasing the average Interaction Set for Checkpointing (I_{CHK}).

Applications	Bar	Cho	Fft	Fmm	Rdx	LuC	LuN	Vol	WSp	WNq	Rad	Oce	Ray	Bla	Flu	Fer	Str	Apa	Avg
% Increase in I_{CHK} due to FPs	1.3	1.4	4.9	1.4	6.4	6.0	4.8	1.4	1.0	0.5	0.0	0.0	0.0	1.6	1.3	1.9	0.6	2.4	2.0
Log Size (MB)	3.0	8.4	15.9	5.0	5.4	11.8	12.5	4.1	0.7	7.5	2.2	29.0	2.4	3.0	5.6	4.7	2.1	6.3	7.2
% Increase in coher. messages	8.4	3.6	3.3	3.0	6.6	3.9	3.1	3.4	4.1	2.5	7.5	3.7	4.8	2.1	4.6	3.2	2.6	4.7	4.2

Table 1: Characterization of *Rebound* for SPLASH-2 (64 processors) and for PARSEC and Apache (24 processors).

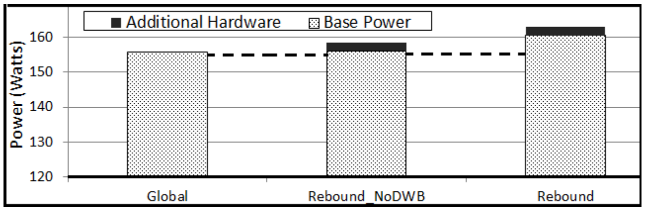


Figure 15: Estimated power consumption for SPLASH-2.

The first row of Table 1 shows the average increase in I_{CHK} due to these false positives. We see that the average increase across all applications is a very small 2.0%.

The second row of Table 1 shows the maximum log space required for a checkpoint interval. It is the maximum number of writebacks during a checkpoint plus the unique writebacks observed until the next checkpoint. On average, it is only 7.2 MB. Finally, the third row of the table is the additional number of messages (over the regular cache coherence protocol) necessary to maintain the LW-ID bits and Dep registers. On average, these messages only increase the number of messages in the machine by 4.2%

7. RELATED WORK

Section 2 already described the most related work. Our work builds on coordinated local checkpointing schemes for shared memory [1, 2, 3, 33]. These schemes, however, only work for bus-based machines. Our work is the first to provide hardware-based coordinated local checkpointing for scalable coherence. The work also builds on shared-memory architectures with high-frequency checkpointing in memory, such as ReVive [21, 23] and SafetyNet [27]. These schemes are global checkpointing schemes, where all processors checkpoint together regardless of their interactions.

The schemes described use Backward Error Recovery (BER). Another approach to recovery is to use Forward Error Recovery (FER) [15]. Unlike in BER, such an approach usually requires hardware replication. Finally, our work does not address the related field of fault detection. There are many fault-detection schemes [20], which trade-off coverage, overhead and cost.

8. DISCUSSION AND FUTURE WORK

We are examining several issues for future work. The first one is adapting Rebound to other directory organizations. In particular, as the number of processors increases, the directory may have pointers to groups (or clusters) of processors. In this case, the MyConsumers/MyProducers registers will be assigned to clusters, and each of their bits will refer to one cluster. Inside a cluster, we can perform global checkpointing.

The design in this paper relies on the coherence hardware to record the inter-thread dependences. In a manycore without hardware cache coherence, the software can generate a graph of the inter-thread communications, to be used by our algorithms to decide which processors to checkpoint or rollback together. The compiler can generate such a graph statically or may emit code that, at runtime, generates it.

Compiler and/or runtime system can enhance Rebound (or variations of it) in many ways. For example, they can selectively enable and disable Rebound for a certain period of time or for a certain range of addresses. They can also compact the footprint of threads or schedule them to reduce the overhead of Rebound.

Finally, we are fleshing out how Rebound interfaces to a highly-efficient storage subsystem based on non-volatile memory.

9. CONCLUSION

Proposed global checkpointing schemes do not scale to upcoming manycores with many tens of processors. To address this problem, this paper presented Rebound, the first hardware-based scheme for coordinated local checkpointing in multiprocessors with directory-based cache coherence. Rebound contributes with several novel features. First, it leverages the transactions of a directory protocol to track inter-thread dependences inexpensively and in a lazy manner. Second, to boost checkpointing efficiency, it introduces novel architectures to: (i) delay the writeback of data to safe memory at checkpoints, (ii) support operation with multiple checkpoints, and (iii) hide checkpointing overhead under barrier synchronization. Third, Rebound introduces distributed software algorithms for checkpointing and rollback sets of processors.

Simulations of parallel programs with up to 64 threads show that Rebound is scalable and has very low overhead. The delayed writebacks at checkpoints and the checkpointing optimization at barrier synchronizations are both very effective, although not additive. During fault-free execution of 64 processors, and without the barrier optimization, Rebound induces an average performance overhead of only 2% — compared to 15% for global checkpointing.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers, the I-ACOMA group members, and Rich Lethin for their comments. This work was supported in part by NSF under grant CCF-1012759; Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC); Sun Microsystems under the UIUC OpenSPARC Center of Excellence; DARPA under UHPC Contract Number HR0011-10-3-0007; and DOE ASCR under Award Number DE-FC02-10ER2599.

11. REFERENCES

[1] R. Ahmed, R. Frazier, and P. Marinos. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In *Int. Symp. on Fault-Tol. Comp. Sys.*, June 1990.

[2] M. Banatre, A. Gefflaut, P. Joubert, C. Morin, and P. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Trans. Comp.*, 45(10), 1996.

[3] M. Banatre and P. Joubert. Cache management in a tightly coupled fault tolerant multiprocessor. In *Int. Symp. on Fault-Tol. Comp. Sys.*, June 1990.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7), 1970.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Int. Symp. on Comp. Arch.*, June 2000.

[6] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Int. Symp. on Comp. Arch.*, June 2006.

[7] T. J. Dell. A white paper on the benefits of Chipkill-correct ECC for PC server main memory. *IBM Microelec. Div.*, Nov 2005.

[8] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comp. Surv.*, 1992.

[9] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. on Comp.*, 41(5), May 1992.

[10] A. Gupta, W. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Int. Conf. on Par. Proc.*, Aug 1990.

[11] Intel Corporation. Single Chip Cloud Computing (SCC) platform overview, Feb 2010. techresearch.intel.com.

[12] B. Janssens and K. Fuchs. The performance of cache-based error recovery in multiprocessors. *IEEE Trans. Par. Dist. Syst.*, 5(10), 1994.

[13] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Int. Symp. on Fault-Tol. Comp.*, June 1995.

[14] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Soft. Eng.*, 1987.

[15] P. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, Inc., 1990.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Prog. Lang. Design and Impl.*, June 2005.

[17] Y. Masubuchi, S. Hoshina, T. Shimada, H. Hirayama, and N. Kato. Fault recovery mechanism for multiprocessor servers. In *Int. Symp. on Fault-Tol. Comp.*, June 1997.

[18] C. Morin, A. Gefflaut, M. Banatre, and A. Kermarrec. COMA: An opportunity for building fault-tolerant scalable shared memory multiprocessors. In *Int. Symp. on Comp. Arch.*, May 1996.

[19] C. Morin, A. Kermarrec, M. Banatre, and A. Gefflaut. An efficient and scalable approach for implementing fault-tolerant DSM architectures. *IEEE Trans. Comp.*, 49(5), 2000.

[20] S. Mukherjee. *Architecture Design for Soft Errors*. Elsevier Inc., Burlington, MA, USA, 2008.

[21] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *Int. Symp. on High-Perf. Comp. Arch.*, Feb 2006.

[22] J. Plank and K. Li. Faster checkpointing with N+1 parity. In *Int. Symp. on Fault-Tol. Comp.*, June 1994.

[23] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Int. Symp. on Comp. Arch.*, May 2002.

[24] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Soft. Eng.*, 1(2), June 1975.

[25] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Jou. of Res. and Dev.*, 52(4/5), 2008.

[26] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, Jan 2005. http://sesc.sourceforge.net.

[27] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Int. Symp. on Comp. Arch.*, May 2002.

[28] F. Sultan, L. Iftode, and T. Nguyen. Scalable fault-tolerant distributed shared memory. In *Int. Conf. on Super.*, 2000.

[29] D. Sunada, M. Flynn, and D. Glasco. Multiprocessor architecture using an audit trail for fault tolerance. In *Int. Symp. on Fault-Tol. Comp.*, June 1999.

[30] D. Tarjan, S. Thoziyoor, , and N. Jouppi. CACTI 4.0. Technical report, HPL-2006-86, HP Laboratories, 2006.

[31] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Int. Sol. State Cir. Conf.*, Feb 2007.

[32] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A memory system simulator. *SIGARCH Comp. Arch. News*, 33(4), 2005.

[33] K. Wu, K. Fuchs, and J. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Trans. Par. Dist. Sys.*, 1(2), 1990.

APPENDIX A: NO DOMINO EFFECT

Assume a maximum fault detection latency L. In Rebound, upon a fault, all the processors in I_{REC} roll back to their most recent safe (i.e., completed before L cycles ago) checkpoint. We now prove: (1) that the set of the most recent safe checkpoints always form a consistent state of the system and (2) that a rollback wastes a bounded time and, therefore, there is no domino effect.

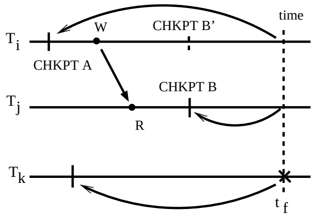


Figure 16: Recovery.

(1) Let thread T_k detect a fault at time t_f. Let T_i and T_j be any pair of threads in T_k's I_{REC}. Rebound rolls them to their most recent safe checkpoints, *CHKPT A* and *CHKPT B*, respectively. Assume that these are an *inconsistent* set of checkpoints. Since they are inconsistent, there must exist a RAW dependence (W to R as shown in Figure 16), such that T_i, on re-execution, re-produces the data that T_j does not re-consume. However, if such a dependence existed, then on the initial run when T_j took its checkpoint *CHKPT B*, it would have forced its producer T_i to checkpoint as well, say *CHKPT B$'$*. As *CHKPT B* is a safe checkpoint for thread T_j, so should be *CHKPT B$'$* for T_i. This contradicts the claim that *CHKPT A* is the most recent safe checkpoint for T_i. In this way, by arguing for any pair of processors at a time, we prove that **the set of the most recent safe checkpoints is always consistent.**

(2) On a fault, any processor at most rolls back to the latest checkpoint that it completed more than L cycles ago. Hence, for any processor, the most recent safe checkpoint is no more than L + *CKPT_INTERVAL* cycles ago. This time bounds the amount of work wasted to recovery by any processor. This proves that **Rebound has no domino effect**.