REBOUND: SCALABLE CHECKPOINTING FOR COHERENT SHARED MEMORY

BY

RISHI AGARWAL

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Josep Torrellas
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.
To my parents, for their love and support.
ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor Prof. Josep Torrellas for his continuous guidance, support and motivation. This work would not have been possible without his counseling and encouragement. I am really thankful that I got an opportunity to work with him and learn from him.

I would also like to express my gratitude for my colleague and friend, Pranav Garg for his help and participation in this project. I also thank my colleague Ulya Karpuzcu and Prof. Zbigniew Kalbarczyk for their insightful comments, questions and suggestions.

Last but not the least, I would like to thank my parents, sister and my friend Pooja Agarwal for their encouragement, patience and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1</td>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>CHAPTER 2</td>
<td>CHECKPOINTS &amp; SHARED MEMORY</td>
</tr>
<tr>
<td>2.1</td>
<td>How Dependences Affect Checkpointing and Rollback</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>REBOUND DESIGN</td>
</tr>
<tr>
<td>3.1</td>
<td>Main Idea</td>
</tr>
<tr>
<td>3.2</td>
<td>Fault Model</td>
</tr>
<tr>
<td>3.3</td>
<td>Basic Operation of Rebound</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>BOOSTING EFFICIENCY &amp; USABILITY</td>
</tr>
<tr>
<td>4.1</td>
<td>Delayed Writeback of Dirty Cache Lines</td>
</tr>
<tr>
<td>4.2</td>
<td>Multiple Checkpoints</td>
</tr>
<tr>
<td>CHAPTER 5</td>
<td>EVALUATION SETUP</td>
</tr>
<tr>
<td>CHAPTER 6</td>
<td>EVALUATION</td>
</tr>
<tr>
<td>6.1</td>
<td>Size of Interaction Set for Checkpointing</td>
</tr>
<tr>
<td>6.2</td>
<td>Overhead During Error-Free Execution</td>
</tr>
<tr>
<td>6.3</td>
<td>Scalability Analysis</td>
</tr>
<tr>
<td>6.4</td>
<td>Estimated Impact of Output I/O</td>
</tr>
<tr>
<td>6.5</td>
<td>Power Analysis</td>
</tr>
<tr>
<td>6.6</td>
<td>Miscellaneous Characterization</td>
</tr>
<tr>
<td>CHAPTER 7</td>
<td>RELATED WORK</td>
</tr>
<tr>
<td>CHAPTER 8</td>
<td>DISCUSSION AND FUTURE WORK</td>
</tr>
<tr>
<td>CHAPTER 9</td>
<td>CONCLUSION</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>NO DOMINO EFFECT</td>
</tr>
<tr>
<td>REFERENCES</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2.1</td>
<td>Rules to checkpoint and roll back under coordinated local checkpointing.</td>
</tr>
<tr>
<td>3.1</td>
<td>Manycore augmented with the base support for Rebound.</td>
</tr>
<tr>
<td>3.2</td>
<td>Recording inter-thread dependences. All memory accesses in (b) are to the same line.</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of the operation of the distributed checkpointing protocol.</td>
</tr>
<tr>
<td>4.1</td>
<td>Support for delayed writebacks (a and b) and multiple checkpoints (c and d).</td>
</tr>
<tr>
<td>4.2</td>
<td>Optimization of the checkpoint at barriers.</td>
</tr>
<tr>
<td>4.3</td>
<td>Simulated system configuration (a) and applications evaluated (b).</td>
</tr>
<tr>
<td>6.1</td>
<td>Average size of the Interaction Set for Checkpointing for PARSEC and Apache for 24-processor runs.</td>
</tr>
<tr>
<td>6.2</td>
<td>Average size of the Interaction Set for Checkpointing for SPLASH-2 for 32-processor runs (a) and 64-processor runs (b).</td>
</tr>
<tr>
<td>6.3</td>
<td>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.</td>
</tr>
<tr>
<td>6.4</td>
<td>Impact of the Barrier optimization on the checkpointing overhead.</td>
</tr>
<tr>
<td>6.5</td>
<td>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.</td>
</tr>
<tr>
<td>6.6</td>
<td>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.</td>
</tr>
<tr>
<td>6.7</td>
<td>Effect of output I/O on the checkpoint interval.</td>
</tr>
<tr>
<td>6.8</td>
<td>Estimated power consumption for SPLASH-2.</td>
</tr>
<tr>
<td>A.1</td>
<td>Recovery.</td>
</tr>
</tbody>
</table>
CHAPTER 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 [1, 2, 3, 4, 5, 6]. 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 [7, 8], 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 [9]. 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 [10]. 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 [11, 12, 13, 14, 15] and, therefore, is not scalable. For example, Banatre et al. [12] 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 evaluate it; Section 7 reviews related work; Section 8 presents a discussion; and Section 9 concludes.
CHECKPOINTS & SHARED MEMORY

Checkpointing and rollback [16] is the most popular approach to recovery. Checkpointing can be Global, where all processors periodically cooperate to create a single checkpoint (e.g., [1, 2, 3, 4, 17, 5, 6]), or Local, where there is no global checkpoint (e.g., [11, 12, 13, 18, 19, 20, 15]). 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 [11, 12, 13, 15]. Otherwise, processors create checkpoints independently, and the system is called Uncoordinated Local [18, 16, 19, 20]. In uncoordinated local checkpointing, rollbacks may risk the domino effect [21], 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., [13, 1, 3, 4, 17, 5, 6]). 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 [11, 13, 15], or stored in place while copying the safe state into a log until the checkpoint [2, 5, 6, 20], or stored in a different address that is mapped to the original program in lieu of the original address [1, 3, 4, 17].

To provide low-overhead checkpointing for shared memory, Banatre et al. proposed the Recoverable Shared Memory (RSM) [12] 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 [12], which are shown in Figure 2.1. Figure 2.1(a) shows an inter-thread dependence. First, as Figure 2.1(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 2.1(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 2.1: Rules to checkpoint and roll back under coordinated local checkpointing.](image-url)
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 [5] 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 [22] or DRAM raiding [23].

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.

Figure 3.1: Manycore augmented with the base support for Rebound.

Figure 3.2: Recording inter-thread dependences. All memory accesses in (b) are to the same line.

3.2 Fault Model

To understand the fault model, Figure 3.1 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 [24]. 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.
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 [25], 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 3.1 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 3.2(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 3.2(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 3.2(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 3.2(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 3.2(a)).

During a checkpoint interval, there may be multiple dependences on a given variable. Figure 3.2(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 [26], 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 [27]). 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 [5]. 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 [5].

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 3.3 describes it with an example, where P1 initiates a checkpoint. Figure 3.3(a) shows the inter-thread dependences and that P4 had checkpointed after providing data. As seen in Figure 3.3(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 3.3(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 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.
Figure 3.3: Example of the operation of the distributed checkpointing protocol.

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 3.3(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 (\textit{Roll?}) to the processors in
its MyConsumers (which in turn send it to their MyConsumers and so on) and waits to receive
\textit{Accept}, \textit{Decline}, or \textit{Busy} messages. It will receive a \textit{Decline} if a consumer processor has recently
performed an independent rollback and, as a result, cleared its own MyProducers; it will receive
a \textit{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 \textit{all} the processors in the
chip to their checkpoints.
CHAPTER 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 4.1(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 4.1(b)).

Figure 4.1: Support for delayed writebacks (a and b) and multiple checkpoints (c and d).

With this approach, since we overlap the writebacks with useful work from the next check-
point 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 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 4.1(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 4.1(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 4.1(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

Figure 4.2: Optimization of the checkpoint at barriers.

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 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.

### 4.2.1 Optimizing Checkpointing at Barriers

Figure 4.2(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 4.2(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 4.2(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 4.2(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 4.2(c), the $I_{CHK}$ of a processor includes itself and the processor that set the flag.

<table>
<thead>
<tr>
<th>Architecture Parameters</th>
<th>Configurations Evaluated</th>
</tr>
</thead>
<tbody>
<tr>
<td>System: Manycore with up to 64 cores</td>
<td>Rebound: Proposed scheme without Barrier optimization.</td>
</tr>
<tr>
<td>Core: single–issue at 1GHz</td>
<td>Rebound_NoDWB: Rebound without the Delayed Writebacks.</td>
</tr>
<tr>
<td>L1: 16KB, 4-way assoc, 32B line</td>
<td>Rebound_Barr: Rebound with the Barrier optimization.</td>
</tr>
<tr>
<td>Hit delay: 2 cycles round trip</td>
<td>Rebound_NoDWB_Barr: Rebound without the Delayed Writebacks and with the Barrier optimization.</td>
</tr>
<tr>
<td>L2: 256KB, 8-way assoc, 32B line</td>
<td>Global: Global checkpointing (baseline).</td>
</tr>
<tr>
<td>Private, write–back, 16–entry MSHR</td>
<td>Global_DWB: Global with the Delayed Writebacks.</td>
</tr>
<tr>
<td>Hit delay: 8 cycles round trip</td>
<td>Checkpointing Parameters</td>
</tr>
<tr>
<td>Miss delay:</td>
<td>Checkpoint interval: 4M instructions (About 5—8 ms)</td>
</tr>
<tr>
<td>To other L2: 60 cycles round trip (avg)</td>
<td>Number of Dep register sets: 4 maximum</td>
</tr>
<tr>
<td>To main memory: 200 cycles round trip</td>
<td>Availability target: &gt; 99.999% available</td>
</tr>
<tr>
<td>Multistage interconnect</td>
<td>Maximum recovery latency: approx. 860 ms</td>
</tr>
<tr>
<td>Directory module: full map</td>
<td></td>
</tr>
<tr>
<td>Memory: 2 channels</td>
<td>DDR2 DRAMs. Data rate 667 MHz</td>
</tr>
<tr>
<td>W_SIG: 1024 bits, similar to Notary’s PBX</td>
<td></td>
</tr>
<tr>
<td>Chip area: 200 sq. mm. at 45nm</td>
<td></td>
</tr>
<tr>
<td>Hierarchical clock tree</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.3: Simulated system configuration (a) and applications evaluated (b).

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 4.2(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.
CHAPTER 5
EVALUATION SETUP

To evaluate Rebound, we built an analysis tool using Pin [28]. The output of Pin is connected to a detailed multi-processor architecture simulator based on SESC [29] that is interfaced to the DRAMsim [30] main memory simulator. We model a manycore with up to 64 cores like the one in Figure 3.1. 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 4.3(a). The simulator has integrated models of power from CACTI [31] and Wattch [32] 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 4.3(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 [5], 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 magnitude 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 4.3(a). Rebound is our proposed local checkpointing scheme of Sections 3 and 4, without the Barrier optimization of Section 4.2.1. 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 3.1. 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.
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 6.1 and 6.2 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 6.2 shows data for 32 and 64-processor runs. The figures show data for $Global$ and $Rebound$.

![Figure 6.1: Average size of the Interaction Set for Checkpointing for PARSEC and Apache for 24-processor runs.](image)

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 Black-scholes 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 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 6.3 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 6.4 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 6.4: Impact of the Barrier optimization on the checkpointing overhead.](image)

**Overhead Breakdown.** To understand the checkpointing overhead, Figure 6.5 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 in-
Figure 6.5: 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.

Figure 6.6: 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. We see that, in Global and Rebound_NoDWB, WBDelay and WBImpalanceDelay 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 codes for 16, 32, and 64 processors. Figure 6.6(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 6.6(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 6.6(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 6.7 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.
6.5 Power Analysis

Figure 6.8 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%.

![Figure 6.8: Estimated power consumption for SPLASH-2.](image)

| Applications | Bar | Cho | Fft | Fmm | Rdx | LatC | LuN | Vol | WSp | WNq | Rad | Oce | Ray | Bla | Flu | Fer | Str | Apa | Avg |
|--------------|-----|-----|-----|-----|-----|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| % Increase in ICHK 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 6.1: Characterization of Rebound for SPLASH-2 (64 processors) and for PARSEC and Apache (24 processors).

6.6 Miscellaneous Characterization

Table 6.1 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 (ICHK). The first row of Table 6.1 shows the average increase in ICHK 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 6.1 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 main-
tain the LW-ID bits and Dep registers. On average, these messages only increase the number of
messages in the machine by 4.2%
Section 2 already described the most related work. Our work builds on coordinated local checkpointing schemes for shared memory [11, 12, 13, 15]. 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 [33, 5] and SafetyNet [6]. 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) [9]. 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 [25], which trade-off coverage, overhead and cost.
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.
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.
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 A.1: 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 A.1), such that \( T_i \), on re-execution, reproduces 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.
REFERENCES


