Appears in Intl. Symp. on Microarchitecture (MICRO), Barcelona, Spain, Nov. 2005

Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors

Meyrem Kırman   Nevin Kırman   José F. Martínez

Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA

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

ABSTRACT

Checkpointed Early Resource Recycling (Cherry) is a recently-proposed micro-architectural technique that aims at improving critical resource utilization by performing aggressive resource recycling decoupled from instruction retirement, using a checkpoint/rollback mechanism to recover from occasional incorrect execution. In this paper, we explore correctness and performance issues that arise when Cherry-enabled processors are used in chip multiprocessor architectures. We propose mechanisms to address cache coherence, memory consistency, and forward progress issues in such environments. We also provide quantitative insight on the performance impact of the Cherry mechanism on parallel processing.

1 INTRODUCTION

Today's out-of-order processors confront the growing discrepancy of processor and memory speed in part by supporting a large number of in-flight instructions. The hope is to overlap useful computation with long-latency memory accesses. Traditionally, such support is realized largely by increasing the size of pertinent resources such as the register file or the load/store queues. Unfortunately, many such resources cannot scale up arbitrarily without adversely affecting the clock cycle, the pipeline depth, or both. As a result, processors are eventually unable to compensate for the speed gap, and performance suffers. This, compounded with rampant power consumption and design cost problems, makes monolithic microprocessor architectures increasingly unappealing.

In light of this upsetting trend, researchers and industry are moving toward multicore solutions that rely on explicit parallelism to attain new levels of performance. Yet the success of these architectures in many fronts still depends critically on the ability to deliver high single-thread performance.

Checkpointed Early Resource Recycling (Cherry) [23] is a recently-proposed micro-architectural technique that seeks to improve critical resource utilization by decoupling resource recycling from instruction retirement, aggressively recycling resources that are not needed to support branch mispredictions and memory replay traps. To support precise exceptions on instructions whose resources have been recycled, Cherry relies on periodic checkpointing of the processor's architectural registers.

Cherry is a promising technique that can *(a)* provide higher levels of performance without unduly enlarging critical microprocessor resources, or *(b)* maintain a certain level of performance while reducing the size of such resources. This allows Cherry to deliver higher single-thread performance in CMPs with relatively small cores. Unfortunately, Cherry (and other recent checkpoint-based microprocessor proposals [5, 8, 11, 20, 23, 24, 30]) do not generally explore integration in chip multiprocessors (CMP). However, if mechanisms like Cherry are to make an impact in these architectures, it is imperative that their integration be addressed.

In this paper, we explore the integration of Cherry-enabled processors in CMP architectures. We propose mechanisms to address cache coherence, memory consistency, and forward progress issues in such an environment. We also provide quantitative insight on the performance impact of Cherry on parallel processing.

This paper is organized as follows: Section 2 provides background information on Cherry. Section 3 discusses architectural extensions to provide correct parallel execution using Cherry-enabled processors. Section 4 describes the experimental setup and presents an evaluation of the performance impact of Cherry on parallel processing. Section 5 discusses related work. Finally, Section 6 presents our conclusions.

2 BACKGROUND: CHERRY

Cherry is a micro-architectural technique that speculatively recycles critical resources such as physical registers and load/store queue entries. A processor enters Cherry mode by taking a checkpoint of the architectural registers. Once in Cherry mode, the processor recycles resources that are not needed to recover from (relatively frequent) branch mispredictions or memory replay traps. To restrict recycling in this way, the ROB entry of the oldest instruction that can suffer from a memory replay trap or a branch misprediction is identified as the *Point of No Return (PNR)* (Figure 1). Instructions older than the PNR constitute the *irreversible set* at each point in time, and only resources associated with such irreversible instructions may be recycled early. Depending on the target resource, the PNR is a function of the oldest address-unresolved load (whose ROB entry is denoted by U_L), address-unresolved store, (ROB entry called U_S), and unresolved branch (U_B).

To support (relatively infrequent) precise exceptions on irreversible instructions, Cherry relies on its checkpoint support. On such an exception, the processor rolls back to the checkpoint and re-executes in non-Cherry mode until the exception is again raised, at which point it can be handled normally. Then (or if the exception does not re-occur) the processor can continue execution in Cherry mode. Because rollbacks are limited to exception handling on irreversible instructions, they are relatively rare. Nevertheless, to limit the penalty of a potential rollback, a Cherry-enabled processor exits Cherry mode and renews its checkpoint periodically. Each period is called a *Cherry Cycle*. To exit Cherry mode,

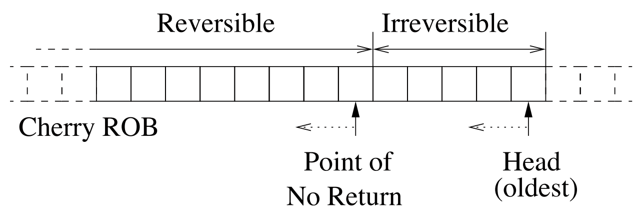

Figure 1: Cherry ROB with Point of No Return (PNR)

a processor freezes the PNR and allows all instructions in the irreversible set to retire. Once the ROB head catches up with the PNR, the processor can renew its checkpoint. We call this process *collapse step*. Exceptions on instructions not in the irreversible set, and all interrupts in general, are handled in this way as well, without requiring a rollback.

2.1 Speculative Memory Updates

Memory updates in Cherry mode must be temporarily buffered so that, in the event of a rollback to the checkpoint, the original memory contents can be recovered by simply discarding such speculative updates. For this purpose, during Cherry execution, memory updates are kept in the processor's local cache hierarchy. Speculatively updated cache lines are marked Volatile (one additional bit per cache line) to distinguish them from nonspeculative ones. Volatile cache lines cannot be evicted to memory. On a processor rollback, all cache lines marked Volatile are gang-invalidated and Volatile bits gang-cleared. On the other hand, on a successful collapse, Volatile bits are gang-cleared, effectively committing all speculatively updated data.

In case a speculative update overwrites nonvolatile dirty data, the data are first written back to memory to ensure availability in the event of a rollback.

Finally, notice that it may be possible that a cache miss find no victims in the corresponding cache set (all lines marked Volatile in that set). This can compromise forward progress if completion of the memory access is needed for the processor to collapse its irreversible set. To minimize the chances of this scenario, a victim cache at the bottom of the local cache hierarchy captures evicted volatile cache lines, and once the number of such cache lines exceeds a certain threshold, the victim cache informs the processor that it ought to renew its checkpoint. In the rare event that the victim cache fills up with Volatile entries and a new replacement cannot take place, the processor rolls back to the checkpoint and re-executes in non-Cherry mode.

2.2 Early Resource Recycling

2.2.1 Load/Store Queue

Modern processors typically incorporate separate load and store queues [10, 34]. These CAM structures hold entries for every in-flight memory instruction in program order, and are used for data forwarding (either load refill or store-to-load forwarding) and, importantly, for enforcing correct memory ordering. Entries are typically allocated at decode/rename, filled with address and data values as these become ready, and released at retirement. In Cherry, however, load and store queue entries are released as soon as they are deemed no longer needed.

Load Queue

Some loads may speculatively issue out of order with respect to previous address-unresolved stores. However, when one such older store later resolves to an overlapping address, the premature load and its dependent instructions are squashed and replayed. This we call a *store-load replay trap*, and the situation is typically detected by inspecting the load queue for matching addresses on resolving a store address. (The second time around, the store queue typically forwards the new value on request.)

Cherry's early load queue recycling is enabled by dividing the load queue into two parts: Load Data Queue (LDQ) and Load Reorder Queue (LRQ). LDQ provides the memory refill functionality of the conventional load queue; LRQ supplies the associative address checking functionality. LDQ entries are shortlived relative to LRQ entries, and thus early recycling in Cherry is applied to the latter. A LRQ entry can be recycled as soon as it is known that the corresponding load is free of replay traps. In a uniprocessor setup that occurs when all its previous stores are address resolved; therefore, PNR for load queue recycling can be defined to be U_S.

Store Queue

Early store queue recycling is achieved by allowing stores to issue to memory ahead of commit, and recycling the corresponding store queue entry. When recycling in this way, it must be known that the previous value in the memory system is not needed anymore. This is guaranteed if (1) no older load is pending address disambiguation, (2) no older load is subject to replay trap, and (3) the store is not subject to squash due to branch misprediction. Based on these three conditions, the PNR for store queue recycling can be defined as oldest(U_L, U_S, U_B). (Notice that recycling is still performed in program order, to ensure in-order updates to the same address.)

2.2.2 Register Recycling

Typically, a physical register holding an architectural value is reclaimed when a subsequent instruction that renames to the corresponding logical register retires. Cherry, on the other hand, recycles physical registers as soon as their values are not needed by subsequent instructions. This happens when (1) the producing and all consuming instructions are executed, free of replay traps and not subject to branch mispredictions, and (2) there is a superseding producing instruction that is also not subject to misspeculation. Accordingly, the PNR for register recycling is defined as oldest(U_S, U_B). (Still, registers superseded by the instructions in the irreversible set are recycled only when it is also known that all consumers have executed. To achieve this, some small per-register bookkeeping support is needed.)

3 CHERRY-MP

Generally speaking, in the absence of any interprocessor communication (for example, when executing independent single-threaded applications), each processor should be able to execute in and out of Cherry mode at its leisure. When sharing of (potentially volatile) data occurs, however, nontrivial correctness and performance issues arise. Furthermore, Cherry's early release of load and store queue entries (and the associated data in the case of stores) poses additional memory consistency challenges.

In this section we present *Cherry-MP*, a mechanism to integrate Cherry in CMP architectures so that it can operate on parallel runs. We address Cherry-MP in the context of a bus-based shared-memory multicore organization. First, we discuss data sharing and cache coherence; then, we deal with issues related to both strict and relaxed memory consistency models.

3.1 Data Sharing

Data sharing in Cherry-equipped CMP architectures is different from conventional architectures in that some of the data held in caches may be volatile—that is, subject to rollback. Consider, for example, a case in which processor P0 requests a data value v produced speculatively by processor P1, which executes in Cherry mode. If P0 also executes in Cherry mode, allowing such an exchange is possible in principle, as both processors are already operating with volatile data. However, to ensure correctness, P0's fate must be bound to P1's from that point on: If P1 later rolls back, so does P0, since it has consumed a value that is now stale. Therefore, P0's copy must be marked volatile as in P1 to be able to invalidate the data on a rollback. Only when P1 commits its volatile state (and thus v) to memory can P0 also collapse and fall back to non-Cherry mode.

An alternative solution is to force the processor receiving the request out of Cherry mode prior to the data exchange. This, however, may undermine Cherry's effectiveness, by shortening the time processors spend in Cherry mode. Worse still, it may cause a deadlock if, for example, the request is by a memory operation in the requester's irreversible set, and the receiver requires data held by the requester to complete its Cherry collapse step (which would now require the requester to collapse as well). In general, deadlock may occur whenever two or more processors trying to collapse form a dependency cycle on irreversible memory operations. These situations generally require nontrivial support for deadlock detection and recovery.

Thus, we opt to introduce hardware support to allow sharing of volatile data. (As we explain later, the required support to allow this is in fact quite modest.) Recall, however, that Cherry-equipped processors may execute in non-Cherry mode for a variety of reasons, most commonly to attend precise exceptions, interrupts, or I/O. Thus, if and when data are shared among multiple processors, a subset of them may be executing in non-Cherry mode. In our example, it is also possible that P0 be in non-Cherry mode at the time of the exchange (with P1 still in Cherry mode). In this case, P0 cannot simply consume v without further precaution: If P1 later rolls back to its checkpoint, P0 will be unable to undo the computation associated with v. It is conceivable that P0 be forced to enter Cherry mode at the time it requests a copy of v from P1, which effectively derives in the scenario described earlier. This, however, is contrary to Cherry's principle of executing in non-Cherry mode in order to guarantee progress in critical situations like the ones mentioned. For this reason, as a general rule, a processor should not be forced into Cherry mode externally. On the other hand, a processor *can* be forced out of Cherry mode, as it happens when, for example, the processor receives an external interrupt (Section 2). Thus, in our example, it may be possible for P0 to read v off P1 if P1 is forced to collapse prior to handing a copy of v over to P0, and thus the exchange occur with both processors in non-Cherry mode.

The case of P0 reading v off P1 with P1 in non-Cherry mode at the time of the exchange (or in Cherry mode, but v updated nonspeculatively before P1 entered Cherry mode) can be handled largely as a conventional access: Indeed, because v is part of the memory's architectural state (not marked Volatile), it can be read in by P0 conventionally. In this case, similarly to Cherry for uniprocessors, even if P0 is in Cherry mode the cache line need not be marked Volatile, as it has not been updated speculatively.

Therefore, generally speaking, the specific data sharing mechanism between two processors must consider whether one or both of them execute in Cherry mode during the time the exchange takes place. This is further complicated by the fact that processors may try to get in and out of Cherry mode, or even rollback during such exchanges. We address all these cases in detail in this section.

Coherence Protocol

A natural choice for a baseline cache coherence protocol on which to build Cherry-MP support is a MOESI protocol [1, 32, 33]. In an n-processor system, MOESI allows several copies of a cache line across processors that are possibly incoherent with the copy in memory. Among those copies, the *Owned (O)* copy is responsible for (1) providing a copy to any new sharer, and (2) writing back the copy if it replaces the cache line. The other copies remain in *Shared (S)* state. Because Cherry requires to keep volatile data off memory, MOESI is convenient in order to share speculatively modified data across processors safely. Table 1 compiles MOESI's states; the rightmost column indicates whether the state is apt to hold volatile data. Notice that, in the case of Shared state, it is only possible to hold volatile data if there is an Owned copy elsewhere in the system—otherwise, the data must be necessarily consistent with memory. This is precisely the final state for v in the earlier example of P0 reading v off P1, with both processors in Cherry mode and v marked volatile in P1: P0 Shared, P1 Owned.

State	**Cache-line's State Description**	**Can be volatile?**
Invalid	Invalid data	No
Shared	Valid data, possibly inconsistent with memory	Yes
Exclusive	Valid data, consistent with memory, present only in one cache	No
Owned	Valid, dirty data, possibly shared	Yes
Modified	Valid, dirty data, present only in one cache	Yes

Table 1: Summary of MOESI protocol states and whether they can hold volatile data.

The extended MOESI state transition diagram to support Cherry-MP is given in Figure 2. By leveraging MOESI's Owned state, we accommodate Cherry's buffering of volatile data without introducing many significant changes to the protocol. We largely follow the labeling convention in [12]. For the sake of brevity, we only discuss the changes introduced to the baseline protocol. They are:

- Speculative writes always mark the writer's cache line Volatile, similarly to the case of Cherry for uniprocessors. Speculative writes on nonvolatile dirty cache lines (Modified or Owned state) must force a writeback of the original contents to main memory, in case a rollback later undoes the speculative update. The speculative write may be initiated by a local processor (identical case to Cherry for uniprocessor), or by a remote

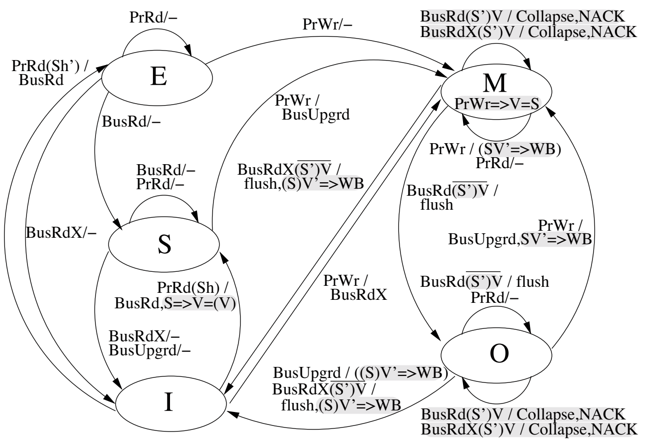

Figure 2: Extended MOESI state transition diagram, where shaded elements are protocol extensions to enable Cherry-MP. Requests that trigger state transitions and any consequent actions appear as *Trigger*/*Action*. Trigger requests may be handled differently depending on certain conditions, and these are shown immediately after the request label. Multiple actions are separated by comma *(X,Y)*, and *(X=>Y)* indicates that Y is performed only if condition X holds. Some abbreviations used: *(S)*: speculative bus request; *(S')*: nonspeculative bus request; *S*: speculative processor request; *(Sh)*: shared snoop result asserted; *(Sh')*: shared snoop result not asserted; *V*: volatile cache line; *V'*: nonvolatile cache line; *(V)*: incoming data is volatile; *WB*: writeback; *flush*: provide data on the bus.

processor through a *BusRdX* (general case) or a *BusUpgrd* (case of request from processor in Shared state to processor in Owned state).

To prompt a writeback in the case of a remote speculative update, the requester must assert on the bus that the access is speculative, lest the current owner may forward the cache line and hand off ownership without updating memory (the default outcome in conventional MOESI). Moreover, as in conventional write-invalidate MOESI, on a remote speculative write request, the original cache line is ultimately invalidated.

On the other hand, speculative writes on volatile dirty cache lines must not result in a write back to main memory in any case. If the cache line is marked Volatile+Dirty elsewhere, the protocol simply forwards the cache line to the requester, if needed, and invalidates the old copy. Notice, however, that if the writer later rolls back, it is not possible to recover the older volatile version of that cache line, and thus the original owner should be rolled back as well. How to bookkeep these and other necessary relations is explained later.

- Speculative reads mark the reader's cache line Volatile, but only if the original copy was a cache line already marked Volatile. This is so that, in the event of a rollback (collapse), all live copies of the speculatively updated value are properly discarded (committed). To support this, on a miss, the supplier of the value must put on the bus its cache line's Volatile bit as part of its snoop response. Notice that, volatile, clean cache lines can be silently dropped by the reader, provided there exists a mechanism to roll back the reader if the producer does so. In the earlier example where P0 reads v off P1, both processors in Cherry mode, a rollback in P1 should drag P0 with it. If at that point P0 still holds a copy of the cache line, it is necessarily marked Volatile, and thus gang-invalidated. The specific mechanism to track the rollback/collapse relations between Cherry producer and consumer processors is explained later.
- A (remote) nonspeculative read request (a *BusRd* or a *BusRdX*) on a cache line marked Volatile (necessarily in either Modified or Owned state, since we restrict cache-to-cache transfers to these) forces the processor receiving the request to collapse. (The requester must assert on the bus that the access is nonspeculative.) This is the case in the earlier example of P0 reading v off P1 with P0 and P1 in non-Cherry and Cherry mode, respectively. Importantly, the request is rejected, with the understanding that the reader will retry after some time (if still interested). This is to allow the owner time to collapse and avoids potential deadlock conditions. Once the processor collapses, it can service the request as a conventional read request on nonvolatile data. (If the requester enters Cherry mode in between retries, handling defaults to the case explained above.) Notice that, if collapse is unsuccessful (i.e., the owner suffers from a rollback), the next attempt by the reader is still able to find the correct copy elsewhere.

As in the case of Cherry for uniprocessors, on a rollback by a processor, any cache line marked Volatile in its local cache hierarchy is gang-invalidated regardless of its MOESI state; on a collapse of a processor, volatile status of all its cache lines marked volatile are gang-cleared. For simplicity, these are not reflected in Figure 2.

3.2 Sharing History

Because sharing of volatile data between processors in Cherry mode is permitted, we ought to make sure that rollback and collapse operations by one processor drag all processors whose sharing with the first one (including false sharing) *binds them* for that particular operation. Given two processors i, j operating in Cherry mode, we say that processor i is bound by rollback (collapse) to processor j, and we denote it by $j \rightarrow i$, if a rollback (collapse) by j should cause i to roll back (collapse) as well. Notice that these relations are directional. In this section we explain all rollback/collapse relations in detail, then describe possible hardware mechanisms to track such relations at runtime.

3.2.1 Rollback and Collapse Relations

On a bus request by processor i, if the controller determines from the snoop responses that the data involved reside Volatile+Dirty in processor j, rollback relations between i and j are formed at that time as follows:

- Case of read request (*BusRd*): One rollback relation is formed: i must roll back if j does so later on, since i has consumed a value that was produced during the computation by j that has now been undone. Therefore, i becomes bound by rollback to j.
- Case of read exclusive request (*BusRdX*): Two rollback relations are formed: (1) i must roll back if j does so later on, since the read-exclusive request ultimately allows i to read any value produced by j in that cache line as the case before; (2) j must roll back if i does so later on, since on a rollback by i the data originally produced

by j cannot be recovered, making it impossible for j to use it or commit it in a subsequent collapse. Therefore, i and j become bound by rollback to each other.

- Case of upgrade request (*BusUpgrd*, data necessarily in Owned state in j): One rollback relation is formed: j must roll back if i does so later on, since on a rollback by i the data originally produced by j cannot be recovered. Notice that relation (1) in the read-exclusive case does not apply here, since i already has read access to an identical copy of the cache line at the time of the request. (If the copy was obtained through a read request with both processors in Cherry mode, then relation (1) was established at that point.) Therefore, only j becomes bound by rollback to i.

Collapse relations are symmetric to rollback relations; they are formed as follows:

- Case of read request (*BusRd*): One collapse relation is formed: j must collapse if i does later on, since i is committing a cache line potentially containing values originally produced speculatively by j. Therefore, j becomes bound by collapse to i.
- Case of read-exclusive request (*BusRdX*): Two collapse relations are formed: (1) j must collapse if i does so later on, since i is committing a cache line with values potentially produced speculatively by j as before; (2) i must collapse if j does so later on, since the read-exclusive by i makes it impossible for j to commit the cache line without i's complicity (the cache line is no longer in j's cache). Therefore, i and j become bound by collapse to each other.
- Case of upgrade request (*BusUpgrd*, data necessarily in Owned state in j): One collapse relation is formed: i must collapse if j does so later on, since a collapse by j requires i's help in committing that cache line. Notice that relation (1) in the read-exclusive case does not apply here, since i already has read access to an identical copy of the cache line at the time of the request. (If the copy was obtained through a read request with both processors in Cherry mode, then relation (1) was established at that point.) Therefore, only i becomes bound by collapse to j.

Overall, directional relations are formed whenever a processor in Cherry mode puts a request on the bus that finds the data marked Volatile+Dirty in another processor, and each type of request derives in different (and always symmetric) rollback and collapse relations. Table 2 summarizes this discussion.

Operation	**Rollback Relation**	**Collapse Relation**
BusRd	Owner → Requester	Requester → Owner
BusRdX	Owner → Requester Requester → Owner	Requester → Owner Owner → Requester
BusUpgrd	Requester → Owner	Owner → Requester

Table 2: Rollback and collapse relations formed as a result of bus requests by a processor in Cherry mode (*Requester*) that finds the data owned speculatively by another processor (*Owner*).

3.2.2 Hardware Support

To successfully implement sharing across Cherry-enabled processors, the hardware must abide by the rollback/collapse relations described above. Perhaps the simplest way to handle sharing history properly in all cases is to assume all-to-all relations at all times. That is, a rollback (collapse) by any processor derives in a global rollback (collapse) by all processors. The main drawback of this approach is that *any* rollback/collapse affects *all* processors, which may impact performance and scalability, as many affected processors may have no sharing history (directly or indirectly) with the initiator.

We would like to track sharing history across processors so that the number of processors involved in a rollback/collapse operation be minimized (within the constraints imposed by the rollback/collapse relations). Unfortunately, the coherence protocol alone is not designed to track sharing over time—e.g., did P0 *ever* read something produced by P1 since the last time they entered Cherry mode? Instead, we opt to add dedicated hardware to track sharing history across processors.

To track sharing history across processors, we extend the bus controller to keep a *sharing history table*. A sharing history table for n processors can keep track of rollback and collapse relations using two $n \times n$ matrices R and C, respectively. Element r_{ij} (c_{ij}) indicates whether processor j should roll back (collapse) following processor i's rollback (collapse). Thus, on a rollback (collapse) of processor i, it is sufficient to index row i of matrix R (C) to find all processors that should also roll back (collapse) as a result.

The real estate to support this sharing history table is modest at one bit per matrix cell, which adds up to a meager 8B per matrix for eight processors, or 128B per matrix for 32 processors. (We later show that these two matrices can in fact fold into one, by exploiting the symmetry of rollback and collapse relations, thus further reducing the history table's overall size.) The main reason for such modest size is that sharing history is kept at the processor level, without regard to the actual amount or purpose of the data exchanged.

R and C matrices are initially set to the unit matrix, which trivially relates each processor with itself. Incorporating a relation $i \rightarrow j$ to matrix R (C) is easily accomplished, by OR-ing row j into each row k for which cell r_{ki} (c_{ki}) is set.

Figure 3 is a four-step example of forming relations among four processors. Assume this to be matrix R, and that all processors are in Cherry mode. In snapshot 1, no relations exist. If, for example, P2 reads data speculatively updated by P1, a rollback by P1 later on should result in P2 rolling back as well, since P2 has consumed an uncommitted value originally produced by P1 that no longer exists. This is recorded in matrix R by OR-ing row 2 into every row k with cell r_{k1} set—only row 1 in this case—, resulting in snapshot 2. Some time later, if for example P3 reads volatile data off P2 (not necessarily the same as before), it becomes bound to roll back if either P2 or (by transitive property) P1 does so. This time row 3 is OR-ed into rows 1 and 2, both of which have their bit for P2 set. The end result is snapshot 3, which not only reflects the P2→P3 relation, but also the P1→P3 formed indirectly. Finally, snapshot 4 shows the matrix contents after P2 reads some volatile data off P4.

Recall from Section 3.2.1 that the formation of collapse relations is symmetric to that of rollback relations in every case. It can be shown that the collapse matrix C is the transpose of matrix R at all times, and thus only one matrix is needed to handle both rollbacks and collapse steps, using row (column) i to identify processors bound by rollback (collapse) to i.

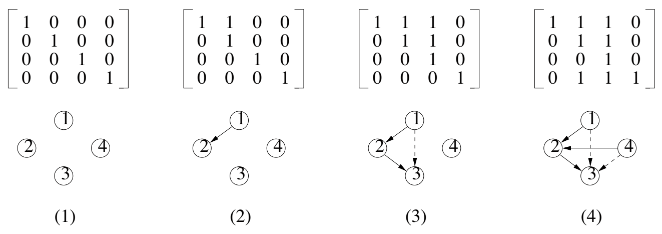

Figure 3: Tracking sharing history: Example of a four-step sequence involving four processors. Solid and dashed arrows represent relations formed directly and indirectly, respectively.

3.2.3 Bounding Rollback Relations

In a large system, as a result of forming rollback relations, a rollback by one processor can cause rollbacks in an elevated number of processors. Consider the extreme case in which all processors are bound by rollback to processor i, and processor i suffers an exception on an irreversible instruction. The ripple effect of this exception would reach *every* processor in the system, and a potentially very large amount of computation would be discarded. To limit the cost of a potential rollback, it is conceivable to introduce mechanisms to bound relations. One possibility is to limit the total number of rollback relations in the system (total number of set bits in the history table). Another possibility is to limit the number of processors that are dependent by rollback on any particular processor (total number of set bits in any one row of the history table).

Whichever the limit imposed, once reached, the system may force a collapse of all or some processors in order to disintegrate some rollback (and collapse) relations. The particular trigger mechanism and collapse actions depend on factors such as the probabilistic distribution of exceptions, the topology formed by rollback relations, and the overhead associated with a collapse step. It is out of the scope of this paper to delve into such details.

3.3 Transition Operations

3.3.1 Entering Cherry Mode

As in the case of Cherry for uniprocessors, we want processors to continue executing uninterrupted when entering Cherry mode. At the time a processor i decides to enter Cherry mode, there may be one or more (depending on the consistency model implementation) unperformed stores by the processor. These stores are necessarily nonspeculative, since they issue to memory from the processor at commit.

If the PNR is allowed to advance at this point, a deadlock situation may arise as follows: One such nonspeculative store may be to a cache line marked Volatile elsewhere in processor j. Per coherence rules (Section 3.1), j is forced to collapse before it can service a nonspeculative request. Suppose, however, that i establishes (possibly indirectly) a collapse relation with j, and is instructed to collapse as a result. Furthermore, assume there is a load operation in i's irreversible set to the same address as the incomplete nonspeculative store (thus waiting at the MSHR). A deadlock cycle then forms as follows: The nonspeculative store cannot be serviced until j collapses; j may not collapse until all processors in its group do so—including i; i may not collapse until the load in its irreversible set completes; and the load may cannot complete until the nonspeculative store brings the data back.

To break this kind of deadlocks, we impose the restriction that processors entering Cherry mode may not initiate early recycling until all outstanding nonspeculative memory operations (stores, since outstanding loads are considered speculative from the point of entering Cherry) complete. This guarantees a zero-size irreversible set in the example before, and thus the irreversible load cannot possibly exist.

3.3.2 Leaving Cherry Mode

Collapsing a processor and the processors that relate to it (obviously related by collapse) is typically a two-step process as follows: When a processor initiates a collapse, it puts a special collapse request on the bus. The bus controller picks up the request and, using the history table's column for that processor, it determines the processor list that relates to this processor by collapse, and instructs those processors to initiate collapse as well.

As processors collapse their irreversible set, they notify the bus controller. Such processors need not stall execution, however they remain in Cherry mode (although with no early recycling taking place) until they are acknowledged by the bus controller. Notice also that, as a result of uninterrupted execution, new relations may be formed, in particular with this group. In that case, the bus controller promptly instructs those new members to collapse as well. Once all processors involved in a group have collapsed their irreversible set, the controller puts a complete-collapse command on the bus, and processors gang-clear their Volatile bits and release their checkpoint, effectively completing the collapse step and falling back to non-Cherry mode. Finally, the bus controller resets the nondiagonal cells of all rows and columns that pertain to the collapsed processors.

The procedure for collective rollback is analogous, except that the processor list is determined by indexing the history table's row for the initiator, and that it is a one step process. Therefore, a rolled-back processor need only wait for the bus controller to clear its row and column to restart execution from the checkpoint.

3.4 Enforcing Memory Consistency

To successfully build multicore hardware out of Cherry-equipped processors, it is important to ensure that Cherry's early recycling and collapse/rollback mechanisms do not interfere with the underlying memory consistency model. In this section we discuss the integration of Cherry in multiprocessors that use both relaxed (e.g., release consistency) and strict (e.g., sequential consistency) memory consistency models.

3.4.1 Relaxed Consistency

Relaxed consistency models [2, 12], such as weak or release consistency [17], distinguish between *ordinary* and *synchronizing* memory operations. Ordinary memory operations to different addresses are generally allowed to perform out of program order. If the programmer or the compiler wishes to impose certain ordering restrictions between memory operations (for example, across boundaries of a critical section), synchronizing memory operations are used. Synchronizing memory operations restrict ordering of memory accesses across them, most notably:

- *Fences* (weak consistency), which do not allow memory operations to perform out of program order with respect to the fence itself.

- *Acquires* (release consistency), which do not allow memory operations after the acquire operation in program order to perform until the acquire itself has performed.
- *Releases* (release consistency), which do not allow the release itself to perform until all memory operations before the release operation in program order have performed.

It is important that entering and leaving Cherry mode (whether by collapse or by rollback) do not affect the correctness of synchronized code. Consider, for example, the case of a processor that acquires a lock in non-Cherry mode, enters Cherry mode within the critical section, and then releases the lock. If the processor now rolls back to the checkpoint, is the execution semantically equivalent to some conventional and correct execution in which the processor executes up to the checkpoint? Fortunately, the answer to this question is Yes. The key lies in the fact that (1) processors in non-Cherry mode cannot consume volatile data (including volatile synchronization variables) until the owner collapses and commits all data, and (2) on a rollback, all volatile-data-dependent processors roll back as well (Section 3.2).

It is also important that early resource recycling do not interfere with the memory consistency model. Aggressive relaxed consistency implementations, based on executing certain memory operations speculatively in violation of the memory consistency mechanism, have been proposed in the past. Most notably, Gharachorloo et al. [16] propose that loads in a critical section execute speculatively ahead of fence or acquire operations, and be replayed if prior to performing the fence/acquire, another processor expresses a conflicting interest in the same data, or the cache line is selected for eviction. In invalidation-based coherence protocols, conflicting accesses by other processors typically result in an invalidation or intervention message sent to the speculative processor, and thus can be detected by performing an associative search on the load queue at the time the invalidation/intervention is received. To support the ability to detect such conflicts, and to replay the affected loads and dependent instructions, in Cherry-equipped processors we must disable early recycling past an incomplete fence or acquire operation. Therefore, in this scenario, we further restrict the PNR to not advance past an incomplete fence or acquire operation.

Likewise, the store queue entry of a fence/release operation should not be recycled until all prior memory operations are complete, since recycling store queue entries in Cherry implies issuing their stores to memory (subject to contention of course). Therefore, the PNR for store queue entries should also not advance past fence/release operations until all prior memory operations are complete.

Load-Load Replay Traps

In relaxed consistency models, a special situation appears when a load executes speculatively ahead of an older load (in program order) whose address is unresolved. If the older load's address later resolves and as a result the accessed location overlaps with that of the speculative load, there is a risk of incorrect execution if a store by another processor, again to an overlapping location, takes place in between. If left uncorrected, the older and newer loads (in program order) end up with the newer and older version of the data, respectively—clearly an incorrect outcome. To resolve this, at the time the address of the older load resolves, the load queue is inspected, and if a conflict is found, the newer load and its dependent instructions are replayed. This we call a *load-load replay trap*. The original Cherry proposal [23] addresses load-load replay traps, by adding U_L to the PNR definitions. Thus, we adopt this additional restriction as well.

3.4.2 Strict Consistency

For the sake of brevity, and without loss of generality, we restrict our discussion to sequential consistency. Sequential consistency [2, 12] is the most conservative memory consistency model. It requires that the result of any execution be the same as if memory operations in each processor performed in strict program order. However, existing processors such as the MIPS R10000 [37] implement an aggressive (i.e., speculative) version of sequential consistency. As in the case of speculative loads past acquire operations in relaxed consistency models [16], loads are issued out of program order without waiting for all previous accesses to complete, but cache replacements and external invalidation/intervention messages are monitored. On an indication that a speculative access is going to result in a violation of sequential consistency, the load operation and dependent instructions are replayed.

The additional requirements to early recycling in this scenario are a generalization of those for relaxed consistency models. We define O_L as the ROB entry of the oldest load for which previous memory operations have not all performed. This corresponds to the oldest load that can be subject to consistency replays. Then, the PNR for early recycling of load queue entries becomes oldest(U_S,O_L). Likewise, the PNR for early recycling (and thus issue to memory) of store queue entries becomes oldest(U_B,U_S,O_L). Stores must perform in order in all cases. Finally, the PNR for early register recycling generally becomes oldest(U_B,U_S,O_L). However, if consistency replays are infrequent, a more aggressive definition of the PNR for register recycling is possible: oldest(U_B,U_S), with the understanding that, should a consistency replay occur in the irreversible set for this PNR, the processor must roll back to the checkpoint, and make sure to re-execute in non-Cherry mode at least past this point of the program execution.

3.5 Multiprogramming Issues

In general, a CMP may support the execution of threads from multiple applications simultaneously, as well as context-switching and thread migration over time. Cherry-MP blends well with such support. Notice that any partitioning of the processors across threads of different applications can be supported by the history table without any modification. This is because threads from different applications generally do not exchange data using the coherence protocol, which is the only way for relations to form in the history table. Thus, at any point in time, the rollback/collapse relations among threads of one application are effectively tracked by the submatrix resulting from deleting the rows and columns that correspond to processors executing threads from other applications. Therefore, no additional support is required.

To support thread scheduling, on a context switch or a thread migration, the processor currently running the thread (and, in the case of thread migration, the destination processor as well) is forced to collapse. Then, the thread may be swapped out or migrated. Recall that, as result of a collapse, all processors bound to this processor must collapse as well. The estimated overhead of this operation, if required to make a scheduling decision, could be derived using the history table. However, we expect this requirement to be rare in most environments. Notice that in no event does any processor require a rollback to its checkpoint as a result of introducing support for multiprogramming.

Finally, our mechanism does not require gang-scheduling of all threads of an application: Since unmapped threads can be considered to be in non-Cherry mode for all practical purposes, there is no need to track relations among them or with mapped threads.

4 EVALUATION

In this section, we conduct a quantitative evaluation of Cherry, by running a set of parallel applications on a number of Cherry-enabled CMP configurations. Our goal is not to conduct a sensitivity study of the possible design choices in implementing Cherry-MP itself, but rather to use a working Cherry-MP implementation to convey some measure of the potential of integrating Cherry in a CMP.

Specifically, we look at the following three scenarios: (1) For a given core design, how does Cherry impact parallel execution time, and how does this impact change as we increase the number of processors? (2) As more transistors become available, if used to build CMPs with larger cores, what is the effect of Cherry on the parallel execution time? (3) Given a transistor budget, how does Cherry impact the performance trade-off between different CMP design points?

The rest of the section is organized as follows: First, we describe the simulation environment and the applications used in this study. Then, we present and discuss the results.

4.1 Simulation Environment

We conduct cycle-accurate execution-driven simulations on detailed models of bus-based, cache-coherent homogeneous CMPs that feature out-of-order superscalar processor cores and a state-of-the-art memory subsystem. Each simulated CMP comprises up to eight cores, and we assume that each application thread runs on a dedicated core. The details of the baseline cores and memory subsystem are shown in Table 3.

Each processor has two levels of on-chip private caches, and they all share a memory-side exclusive L3 cache that is accessed in parallel to main memory. We use a modified MOESI coherence protocol (Section 3.1) and enforce release consistency. Processors communicate and access memory through a shared, split-transaction system bus that runs at half the processor speed and does not allow conflicting outstanding bus requests. All latencies and resource occupancies are modeled in detail.

We explore cores of three different sizes; architectural parameters that vary across cores are shown in separate columns in Table 3. We calculate the load/store queue lookup and register-file access delays, as well as all cache latencies, using CACTI 3.2 [27] on a 90nm process technology. Making use of the results and technology scaling trends presented elsewhere [3, 13, 25], we estimate the rename/wakeup/scheduling delays. We use the same clock frequency in all configurations, however pipeline depth varies.

Following common practice for SPLASH-2, we scale down cache sizes to account for the reduced nature of the input sets provided in the benchmark suite [35]; however, we maintain the latencies of the more realistically sized caches.

We implement Cherry-MP in detail. We do not inject exceptions, however we do force processors to collapse and renew their checkpoint periodically (baseline period is 5μs [23]), or as dictated by Cherry-MP's collapse relations at each point in time.

Processor Cores	**2-issue**	**4-issue**	**6-issue**
Frequency	3.2 GHz	3.2 GHz	3.2 GHz
Fetch/issue/commit width	2/2/3	4/4/6	6/6/9
Inst. window [(Int+Mem)/FP]	32/24	64/48	96/64
ROB entries	96	192	256
Int/FP registers	64/64	96/96	128/128
Int ALUs	2	4	6
Branch units	1	2	3
Int Mul/Div units	1/1	2/2	3/3
FP ALUs	2	3	5
FP Mul/Div units	1/1	2/2	4/4
Ld/St units	1/1	2/2	2/2
Ld/St queue entries	12/12	24/24	32/32
Branch penalty (cycles)	10 (min.)	10 (min.)	11 (min.)
Store forward delay (cycles)	2	3	3
Branch predictor, (Hybrid of GAg + Bimodal)	8K-entry, 13b GHR	16K-entry, 14b GHR	32K-entry, 15b GHR
BTB size	1024	2048	4096
RAS entries	16	24	32
Memory Subsystem	**2-issue**	**4-issue**	**6-issue**
L1 Cache size	16K	32K	64K
L1 Cache size for SPLASH-2	8K	16K	32K
L1 Cache RT	3clk	3clk	3clk
L1 MSHR entries	8	16	24
L2 Cache size	256K	512K	1MB
L2 Cache size for SPLASH-2	32K	64K	128K
L2 Cache RT	10clk	11clk	13clk
L2 MSHR entries	8	16	24
L3 Cache size	8MB	8MB	8MB
L3 Cache size for SPLASH-2	2MB	2MB	2MB
L3 Cache RT	58clk	59clk	61clk
Writeback/Replacement policy	WT/LRU L1, WB/LRU L2, L3		
Cache associativity	4-way L1, 8-way L2, 16-way L3		
Block size	64 bytes		
Cache ports	2 L1, 2 L2, 1 L3		
System bus	256 bits, 1/2 processor speed		
Max. outstanding bus requests	96		
Memory	8-channel Direct Rambus		
Memory bus bandwidth	12.8 GB/s		
Memory access time	161clk on page hit 289clk on page miss		

Table 3: Summary of the baseline hardware modeled in this study. In the table, GHR, MSHR, RAS, and RT stand for global history register, miss status holding register, return address stack, and minimum round-trip time from the processor, respectively. Cycle counts (clk) refer to processor cycles.

4.2 Applications

To evaluate Cherry, we use two and ten applications from the SPEC OMP [6] and the SPLASH-2 [35] application suites, respectively. The application list and simulated problem sizes are given in Table 4. We use MIPS binaries compiled at -O3 optimization level, and run all applications to completion after skipping initialization.

4.3 Cherry Performance and Scalability

Our first experiment seeks to quantify the benefit of using Cherry on a particular CMP architecture, and the variation of Cherry's impact across CMPs with different transistor budgets. (For simplicity, we use the same process technology across all configurations.)

Each plot in Figure 4 shows, for every application under study, the speedup obtained on a Cherry-enabled CMP (*Cherry*) for one, two, four, and eight processors, relative to the performance of a CMP with no Cherry support (*Baseline*) and the same number of processors. From top to bottom, the

Appl.	Description	Problem size
Barnes	Evolution of galaxies	16k particles
Cholesky	Cholesky factorization kernel	tk29.O
FFT	FFT kernel	64k points
LU	LU kernel	512×512 matrix, 16×16 blocks
Ocean	Ocean movements	258×258 ocean
Radiosity	Iterative hierarchical diffuse radiosity method	-room -ae 5000.0 -en 0.05 -bf 0.1
Radix	Integer radix sort kernel	radix 32, 1M integers
Raytrace	3-D ray tracing	car
Water-NSq	Forces and potentials of water molecules	512 molecules
Water-Sp	Forces and potentials of water molecules	512 molecules
Equake	Earthquake modeling	MinneSpec-Large [21]
SWIM	Shallow water modeling	MinneSpec-Large

Table 4: Applications and simulated problem sizes.

plots correspond to experiments done using CMP configurations with two-, four-, and six-issue cores (Table 3).

As a reference point, we also provide the speedups obtained by increasing register file, load queue, and store queue sizes in the baseline by 16 entries each (*Base+16*), 32 entries each (*Base+32*), and by as much as needed to remove the limitations of these resources (*Unlimited*). These enlarged configurations are useful to give an idea of how critical such resources are for each application. For those applications where the performance gap is significant, the quality of Cherry can be quantified by how its speedup compares to those of the enlarged configurations.

The results show across-the-board speedups for Cherry-equipped CMP configurations. Specifically, Cherry achieves 1.13, 1.12, 1.12, and 1.10 average speedups for one, two, four, and eight processors, respectively on the CMPs with two-issue cores. In a few applications, the resource demand decreases as the number of processors scales up, largely due to more memory and synchronization overheads. Accordingly, the speedups obtained using Cherry in these cases decrease as well. The majority, however, exhibit sustained speedups.

As we move to CMPs with four- and six-issue cores (Figure 4*(b)* and 4*(c)*), where the pressure on resources is higher, Cherry is generally more beneficial. For example, average speedup results in the CMP configuration with four-issue cores are 1.16, 1.16, 1.14, and 1.11 for one, two, four, and eight processors, respectively. (Results for six-issue configurations are similar.)

When looking at "resource-hungry" applications, the results are even more encouraging. In the CMP configuration with six-issue cores, for example, we may consider an application to be resource-hungry if its single-processor run experiences a 1.15 speedup or higher in the Unlimited configuration. All but four applications (Radix, Raytrace, Water-NSquared, and Water-Spatial) fall into this category. Their average speedups for the Cherry runs are 1.23, 1.22, 1.19, and 1.14 for one, two, four, and eight processors, respectively. These results are within five percent of the average speedups for the corresponding Unlimited configuration in all cases, thus highlighting Cherry's effectiveness in improving resource utilization.

However, we notice that Barnes (one of the resource-hungry applications), despite delivering good speedups for the Cherry configuration, falls quite short of the performance achieved by the enlarged configurations. Our simulation data shows that Barnes indeed exhibits high load and store queue resource demand. However, while Cherry finds plenty of opportunities to reduce the pressure on the load queue (thus delivering some speedup), it generally fails to produce significant savings on the store queue for this application. Further scrutiny of Barnes's run-time behavior reveals that Cherry's in-order store queue recycling policy (Section 2.2.1) hinders store queue recycling for this application quite significantly, despite achieving an acceptable irreversible set (Table 5).

Overall, as more transistors become available, we expect Cherry in CMPs to maintain or even increase its performance impact.

Appl.	Used ROB (%)	Irreversible Set (% of Used ROB)			Cherry Init Step (Cycles)	Collapse Step (Cycles)
		Reg	LQ	SQ		
Barnes	56.06	29.23	77.87	29.23	65.23	87.37
Cholesky	70.63	80.04	82.14	80.04	364.52	157.4
Equake	74.49	44.34	78.66	44.34	32.58	105.22
FFT	60.97	74.36	83.63	74.36	184.7	329.71
LU	79.84	51.13	62.11	51.13	60.82	66.51
Ocean	69.88	68.49	98.36	68.49	1010.19	606.81
Radiosity	48.82	26.95	77.64	26.95	57.81	57.71
Radix	46.12	13.19	16.61	13.19	226.75	160.79
Raytrace	39.32	16.66	51.35	16.66	26.88	110.38
Swim	96.71	95.59	96.16	95.59	421.72	476.9
Water-NSq	45.81	17.76	56.68	17.76	41.88	32.35
Water-Sp	49.65	15.14	53.46	15.14	37.02	32.56
Average	61.53	44.41	69.56	44.41	210.84	185.31

Table 5: Cherry-MP statistics taken from four-processor runs on the CMP configuration with six-issue cores (Table 3).

4.4 Area-Constrained Scalability

Given a limited transistor budget, one can design a CMP with a few large cores, or with more, smaller cores. Whether the configuration with more cores is preferable to run a particular parallel application depends on whether the additional thread parallelism (more cores) is able to compensate for the loss in single-thread performance (smaller cores). In this experiment, we try to assess the capacity of Cherry to compensate for the loss in single-thread performance as a result of choosing more, smaller cores.

For each of three different scaling scenarios (N=1→2 processors, N=2→4 processors, and N=4→8 processors), we conduct two sets of experiments, each comparing three different configurations that we reasonably assume to be area-equivalent, by reducing the size of each core (Table 3) as we increase N: (1) A CMP configuration with N six-issue (four-issue) cores and no Cherry support (*Baseline*). (2) A CMP configuration with 2N four-issue (two-issue) cores and still no Cherry support (*Constrained*). (3) A CMP configuration with 2N four-issue (two-issue) Cherry-equipped cores (*Constrained-Cherry*).

Furthermore, to provide an optimistic "performance target" in each case, we also simulate an area-unconstrained CMP configuration with 2N six-issue (four-issue) cores—that is, twice the number of cores of Baseline but no core size reduction—and no Cherry support (*Unconstrained*).

Figure 5 shows, for each design point, the speedups of Constrained, Constrained-Cherry, and Unconstrained, relative to the corresponding Baseline.

The plots show that Constrained configurations outperform Baseline in almost all cases. This is because practically all the applications under study exhibit enough parallel efficiency to compensate for the loss in single-thread performance. Nevertheless, this loss is severe enough to create a performance gap with the corresponding Unconstrained configuration. This is

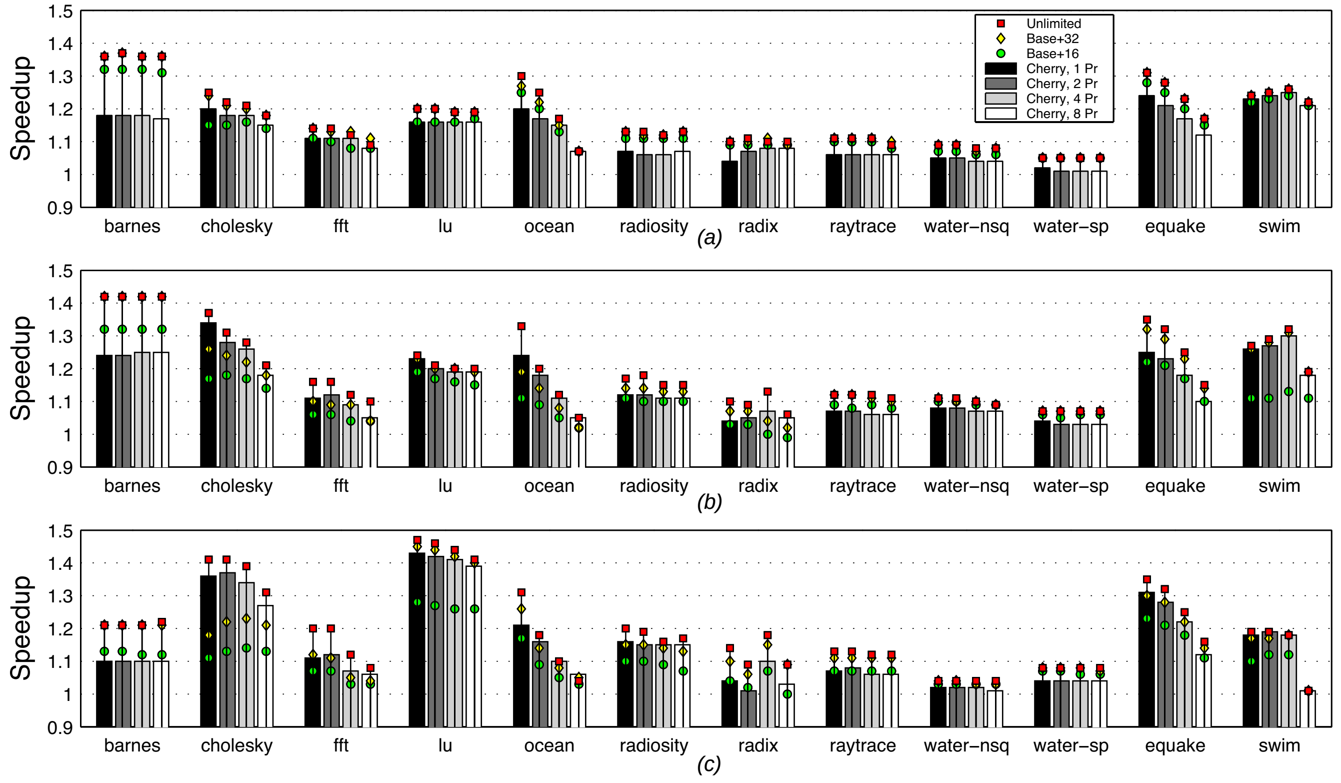

Figure 4: Speedups of Cherry and enlarged Baseline configurations, relative to Baseline for one, two, four, and eight processors. Results are shown for three different core types: two- *(a)*, four- *(b)*, and six-issue *(c)* configurations (Table 3).

particularly significant in the experiment with smaller cores (Figure5*(a)*), indicating that the single-thread performance loss of shrinking from a four- to a two-issue core is noticeably greater than the loss resulting from downsizing from a six- to a four-issue core (which our simulation data confirms).

The same trend shows in the Constrained-Cherry configurations. In the experiments with smaller cores (Figure 5*(a)*), although Cherry improves resource utilization—in fact coming close to the optimum (Figure 4)—the opportunity for speedups in two-issue processors through better resource utilization is not enough to close the significant performance gap between two- and four-issue core configurations. Consequently, although the additional speedups from adding Cherry support bring performance closer to our optimistic performance target, a noticeable gap remains in most applications.

In the experiments with larger cores, however (Figure 5*(b)*), Constrained-Cherry configurations match or even outperform the corresponding Unconstrained configuration in most cases. This indicates that, when using slightly more aggressive cores (but still area-constrained), adding Cherry can effectively deliver the thread-level performance gains of doubling the number of cores without giving up on single-thread performance.

Overall, our area-constrained results show that Cherry is helpful in simultaneously moving toward more, leaner cores, and improving resource utilization in each core, ultimately enabling these architectures to reap the best of both worlds: higher thread-level parallelism (more cores) *and* instruction-level parallelism (smaller, but more efficient cores).

5 RELATED WORK

Several micro-architectural mechanisms based on selective checkpointing [5, 8, 11, 20, 23, 24, 30] have been proposed recently to improve uniprocessor performance; however, they do not generally address integration in CMP architectures. Galluzzi et al. [15], inspired by the potential of kilo-instruction processors [11], present a limit study of the performance impact of using kilo-instruction-sized processors on CC-NUMA multiprocessors. However, they do not address the architectural issues of incorporating checkpoint-based processors to a multiprocessor environment.

Backward error recovery (BER) schemes developed for fault tolerance in general, and targeting shared-memory multiprocessors in particular [4, 7, 19, 26, 29, 36], also employ a checkpointing and recovery mechanism. BER is an error-recovery technique that periodically saves the state of the system and restores it on error detection. Besides serving for different purposes, there are functional differences between these and Cherry-MP.

BER schemes continuously run in checkpointed execution. On the other hand, in Cherry-MP, some processors may execute in non-Cherry mode at any point in time. This requires separate checkpoint allocation and commit processes, as well as disallowing speculative data transfers from Cherry to non-Cherry executions. Another difference is that Cherry-MP faces specific consistency issues due to early recycling of resources, whereas BER schemes generally do not face such issues. Furthermore, the collapse step in Cherry requires additional provisions to avoid deadlock/livelock scenarios.

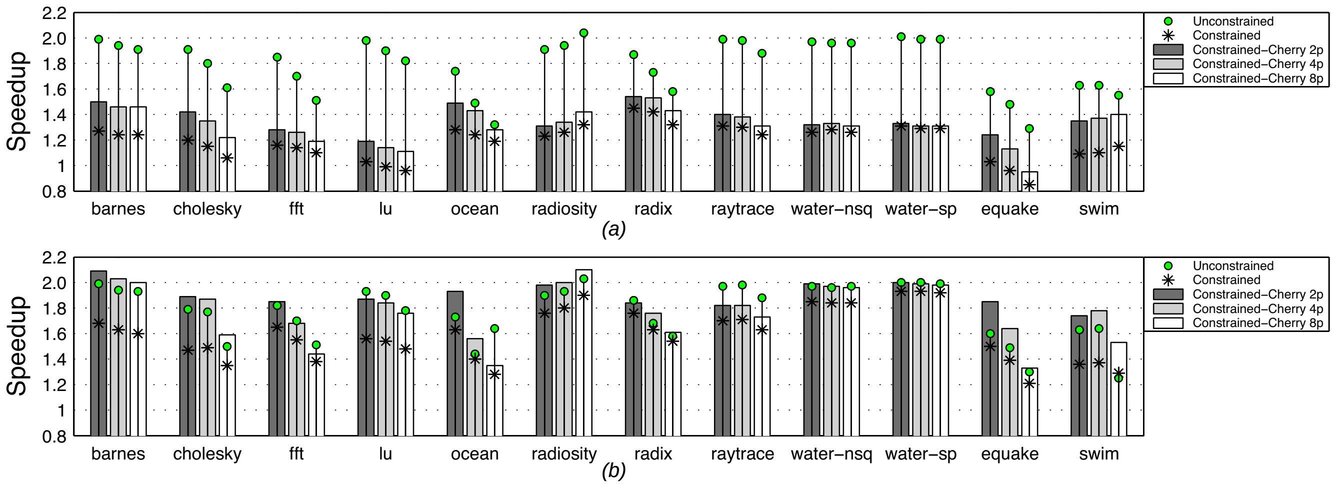

Figure 5: Speedups obtained when doubling the number of cores in three area-constrained scenarios: N=1→2 processors (left bars in each group), N=2→4 processors (middle bars), and N=4→8 processors (right bars). To compensate for the increase in N, cores are shrunk *(a)* from four- to two-issue and *(b)* from six- to four-issue sizes (Table 3).

Among BER schemes, the work by Banâtre et al. [7] proposes the use of groups to involve the minimum possible number of processors on a checkpoint commit or recovery event. Groups are formed by tracking data sharing. Our proposed Cherry-MP mechanism also relies on group formation. However, there are significant conceptual and implementation differences. Apart from the several fundamental differences between Cherry-MP and BER, Cherry-MP differs from Banâtre et al.'s mechanism [7] in that speculative data is not allowed to update main memory, and thus requires no actions in main memory on a recovery event. However, in the BER scheme [7], speculative (current) and recovery data are kept in separate regions of main memory, which requires copying back the recovery data to the current-data region on a rollback. Furthermore, in Cherry-MP, collapsing a group is performed without requiring the processors and caches to block execution, whereas this BER scheme requires that all processors in the committing group flush their dirty data to memory, blocking execution of the processors and caches.

There are memory system proposals that also hold and manage speculative data in the caches [14, 18, 22]. However, they are aimed for TLS-like execution models [9, 28, 31]. Unlike the conventional parallel execution model where threads are nonspeculative and there is a single correct version of a memory location at a time, these TLS-like execution models have speculative threads extracted from a sequential program that run on different processors, and there are multiple correct versions of a memory location at a time. Therefore, while their problem is to provide a thread with a correct version and to establish sequential program memory ordering, our work tries to establish coherent sharing of speculative (volatile) data across nonspeculative parallel threads. As a result, these techniques are not applicable to our original problem (although some of the hardware support may be).

6 CONCLUDING REMARKS

In this paper we have presented Cherry-MP, a mechanism to integrate Checkpointed Early Resource Recycling (Cherry) [23] in bus-based, shared-memory chip multiprocessors (CMPs). Our proposed mechanism comprises three main components: (1) modest extensions to a MOESI coherence protocol to support speculative-data sharing across Cherry-equipped processors; (2) a sharing history table that accurately and efficiently tracks sharing history among processors operating in Cherry mode, which is used to minimize the impact of a rollback/collapse by one processor on others; and (3) small changes to Cherry's early recycling mechanisms in order to comply with the underlying memory consistency model (whether relaxed or strict). The resulting hardware is modest in size, as is the added complexity, and allows correct and efficient integration of Cherry in CMP architectures.

We have conducted detailed simulation-based studies that reassert the efficacy of Cherry in CMPs. Our results show that Cherry-MP is an effective mechanism to increase performance in homogeneous chip multiprocessors composed of either leaner or larger cores, delivering across-the-board speedups for a variety of parallel applications and scaling points.

Furthermore, as more transistors become available and we are able to build same-scale CMPs with larger cores, Cherry exploits the extra opportunity that larger, more aggressive cores provide, delivering higher speedups.

Moreover, when trading off CMP scale and core sizes, configurations with more, leaner cores can use Cherry to recoup part of the "lost" instruction-level parallelism per core, and at the same time reap the benefits of being able to execute the parallel application at a higher scale.

Overall, our proposed Cherry-MP mechanism has the potential to provide a timely and beneficial shift in the architectural trade-offs in current and upcoming CMP designs. By utilizing processor resources more efficiently, Cherry-MP can allow a smoother ride of Moore's law on the CMP front, enjoying better instruction- *and* thread-level parallelism.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback. This work was supported in part by NSF award CCF-0429922 and gifts from Intel. Nevin Kırman was supported in part by an Intel Research Foundation graduate fellowship. Some of the simulations were conducted using computing resources at the National Center for Supercomputing Applications (NCSA).

REFERENCES

[1] Advanced Micro Devices, Sunnyvale, CA. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, February 2005.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical report, Western Research Laboratory, September 1995.

[3] V. Agarwal, S. W. Keckler, and D. Burger. The effect of technology scaling on microarchitectural structures. Technical Report TR2000-02, The University of Texas at Austin, August 2000.

[4] R. E. Ahmed, R. C. Frazier, and P. N. Marinos. Cache-aided rollback error recovery (CARER) algorithms for shared-memory multiprocessor systems. In *20th International Symposium on Fault-Tolerant Computing Systems*, pages 82–88, Newcastle Upon Tyne, UK, June 1990.

[5] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *International Symposium on Microarchitecture*, pages 423–434, San Diego, CA, December 2003.

[6] V. Aslot and R. Eigenmann. Quantitative performance analysis of the SPEC OMPM2001 benchmarks. *Scientific Programming*, 11(2):105–124, 2003.

[7] M. Banâtre, A. Gefflaut, P. Joubert, C. Morin, and P. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Transactions on Computers*, 45(10):1101–1115, October 1996.

[8] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 misses with checkpoint-assisted value prediction. *IEEE Computer Architecture Letters*, 3, December 2004.

[9] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 13–24, Vancouver, Canada, June 2000.

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

[11] A. Cristal, O. J. Santana, M. Valero, and J. F. Martínez. Toward Kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[12] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[13] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *International Symposium on High-Performance Computer Architecture*, pages 40–51, San Jose, CA, February 1996.

[14] M. Franklin and G. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[15] M. Galluzzi, V. Puente, A. Cristal, R. Beivide, J. A. Gregorio, and M. Valero. A first glance at Kilo-instruction based multiprocessors. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 212–221, Ischia, Italy, April 2004.

[16] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *International Conference on Parallel Processing*, pages I355–I364, St. Charles, IL, August 1991.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, Seattle, WA, June 1990.

[18] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *International Symposium on High-Performance Computer Architecture*, pages 195–205, Las Vegas, NV, January–February 1998.

[19] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *13th Symposium on Reliable Distributed Systems*, pages 42–51, Dana Point, CA, October 1994.

[20] N. Kırman, M. Kırman, M. Chaudhuri, and J. F. Martínez. Checkpointed early load retirement. In *International Symposium on High-Performance Computer Architecture*, San Francisco, CA, February 2005.

[21] A. KleinOsowski and D. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1, June 2002.

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

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

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

[25] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-1996-1328, University of Wisconsin-Madison and Digital Equipment Corporation, 1996.

[26] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 111–122, Anchorage, AK, May 2002.

[27] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

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

[29] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, pages 123–134, Anchorage, AK, May 2002.

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

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

[32] Sun Microsystems. *UltraSPARC User's Manual*, July 1997.

[33] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *International Symposium on Computer Architecture*, pages 414–423, Tokio, Japan, June 1986.

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

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[36] K. L. Wu, W. K. Fuchs, and J. H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, April 1990.

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