Appears in the Proceedings of the International Parallel and Distributed Processing Symposium
Santa Fe, New Mexico, April 27-29, 2004

Using Speculation to Simplify Multiprocessor Design

Daniel J. Sorin[1], Milo M. K. Martin[2], Mark D. Hill[3], David A. Wood[3]

[1]Dept. of Elec. & Comp. Engineering
Duke University
sorin@ee.duke.edu

[2]Dept. of Comp. & Information Science
University of Pennsylvania
milom@cis.upenn.edu

[3]Computer Sciences Dept.
University of Wisconsin—Madison
{markhill,david}@cs.wisc.edu

Abstract

Modern multiprocessors are complex systems that often require years to design and verify. A significant factor is that engineers must allocate a disproportionate share of their effort to ensure that rare corner-case events behave correctly. This paper proposes using "speculation for simplicity" to enable designers to focus on common-case scenarios. Our approach is to speculate that rare events will not occur and rely on an efficient recovery mechanism to undo the effects of mis-speculations.

We illustrate the potential of speculation to simplify multiprocessor design with three examples. First, we simplify the design of a directory cache coherence protocol by speculatively relying on point-to-point ordering of messages in an adaptively routed interconnection network. Second, we simplify a snooping cache coherence protocol by treating a rare coherence state transition as a mis-speculation. Third, we simplify interconnection network design by removing the virtual channels and then recovering from deadlocks when they occur.

Experiments with full-system simulation and commercial workloads show that speculation is a viable approach for simplifying system design. Systems can incur as many as ten recoveries per second due to mis-speculations without significantly degrading performance, and our speculatively simplified designs incur far fewer recoveries.

1 Introduction

Shared memory multiprocessors are complicated systems that are difficult to design. Verifying that these designs are correct is even more difficult. As one example, cache coherence protocols are prone to infrequent timing races that exercise difficult-to-test corner cases. As another example, deadlock avoidance presents a challenge in the design of a multiprocessor memory system. The standard solution avoids deadlock in the interconnection network by using virtual channels [7], which increases design and verification complexity. However, even without virtual channels, deadlock occurs rarely. We would like to be able to speculate that rare scenarios, such as corner cases in cache coherence protocols and deadlocks in interconnects, will not occur and recover when they do.

Designers have already discovered the potential of speculation to simplify design, but they have only applied it within the processor core. The key has been to leverage the existing mis-speculation recovery mechanism in the dynamically scheduled core. Several processors resort to a pipeline squash on rare, complicated instructions, such as the Intel Pentium Pro's manipulation of control registers [11]. Similarly, the Pentium 4 uses recovery to handle corner case deadlocks in the scheduler [5]. These processors are speculating that certain instructions or races are rare.

The enabling technology for speculation is fast and efficient recovery; otherwise, even infrequent mis-speculations will unacceptably degrade performance. Modern dynamically scheduled processors provide such mechanisms in the uniprocessor core, allowing speculation *within* a single thread of execution. More recently, there have been several proposals for fast and efficient system-wide checkpoint/recovery of multiprocessors in hardware [19, 17]. These mechanisms capture a consistent global state, allowing speculation *between* multiple processors. Future multiprocessor systems will likely use such system-wide checkpoint/recovery to tolerate the increasing frequency of transient hardware faults in emerging sub-micron technologies. We seek to exploit such a mechanism to simplify the design and verification of multiprocessors.

In this paper, we propose using "speculation for simplicity" to simplify *multiprocessor* design. The cornerstone of our philosophy is to allocate design and verification effort towards common-case, performance-

This work is supported in part by the National Science Foundation (grants: CCR-0309164, CCR-0324878, EIA-9971256, EIA-0205286, and CCR-0105721), an Intel Graduate Fellowship and a Warren Faculty Scholarship (Sorin), a Norm Koo Graduate Fellowship and an IBM Graduate Fellowship (Martin), two Wisconsin Romnes Fellowships (Hill and Wood), Spanish Secretaría de Estado de Educación y Universidades (Hill sabbatical), and donations from Compaq Computer Corporation, Intel Corporation, IBM, and Sun Microsystems. Profs. Hill and Wood have significant financial interests in Sun Microsystems.

Table 1. Using the framework to characterize three speculative designs

	Applications of Speculation for Simplicity		
	Simplify directory protocol by speculating on point-to-point ordering (Section 3.1)	**Simplify snooping protocol** by treating corner case transition as error (Section 3.2)	**Simplify interconnection network** by removing virtual channel flow control (Section 4)
(1) Infrequency of mis-speculation	re-orderings are rare and most re-orderings do not matter	writebacks do not often race with requests to write the block	worst-case buffering requirements are rarely needed in practice
(2) Detection	one specific invalid transition in protocol controller	one specific invalid transition in protocol controller	timeout on cache coherence transaction
(3) Recovery	SafetyNet	SafetyNet	SafetyNet
(4) Forward Progress	selectively disable adaptive routing during re-execution	slow-start execution after recovery	slow-start execution after recovery, with sufficient buffering during slow-start
Result	simpler protocol with rare mis-speculations	protocol almost never exercises corner case in practice	simpler network incurs no deadlocks in practice

critical events rather than rare corner-case events. Such a system may predict that it will not encounter a rare event and speculatively execute based on that assumption. If the system detects mis-speculation (i.e., the rare event occurred), it recovers to a consistent pre-speculation state and resumes execution. Speculation can help in situations in which (a) the design complexity to handle an infrequent event is far worse than that for the common case, (b) detecting the infrequent event is much easier than handling it, and (c) the event is infrequent enough that recoveries negligibly impact performance.

The primary contribution of this work is a framework for simplifying multiprocessor design with speculation. This framework specifies four features necessary for supporting speculation for simplicity: (1) infrequency of mis-speculation, (2) detection of all mis-speculations, (3) recovery from mis-speculation, and (4) guaranteed forward progress. We present this framework in more detail in Section 2.

We develop three concrete examples to illustrate how we use this framework to simplify the two most complicated parts of multiprocessor system design: the cache coherence protocol and the interconnection network. In Section 3, we show how to simplify coherence protocols, with an example of a directory protocol and a snooping protocol. In Section 4, we show how to simplify the interconnection network by removing the virtual channels. While these examples are neither exhaustive nor applicable to all multiprocessors, they do reveal the potential to simplify system designs.

In Section 5, we evaluate our speculative designs with full-system simulation and commercial workloads. Results show that speculation is a viable technique for simplifying system design. In general, a speculative system can maintain its performance even when ten recoveries per second occur, and our speculative systems incur recoveries less frequently than that.

In Section 6, we discuss related work in design simplification, before concluding in Section 7.

2 Framework for Speculation for Simplicity

Our framework specifies the four features necessary for supporting speculation for simplicity. Without these features, speculation for simplicity is not viable.

(1) Infrequency of mis-speculation. Mis-speculation must occur sufficiently rarely that the performance overhead of recoveries is not prohibitive.

(2) Detection of mis-speculations. The system must detect all mis-speculations, and detection mechanisms must not be so complex as to offset gains from speculation.

(3) Recovery. To recover from mis-speculation, we need a recovery mechanism that (a) incurs low runtime overhead during mis-speculation-free execution and (b) can recover the system to a pre-speculation state. While such a scheme may be too expensive to implement strictly for purposes of speculation for simplicity, we can leverage the same recovery mechanism used to improve system availability. For all three of our examples, we use SafetyNet [19], a recently developed hardware mechanism for recovering the state of a multiprocessor system, although other schemes exist [17]. Since our speculation extends outside the processor core, the recovery scheme must encompass the entire memory system, including the caches, cache coherence state, and the memory. Periodically, SafetyNet logically checkpoints the state of the shared memory system and, on mis-speculation, it allows the system to recover to a previous checkpoint. Checkpoints can span hundreds of thousands

of cycles and tolerate long detection latencies. SafetyNet efficiently checkpoints the multiprocessor state, using only hardware, by incrementally logging changes to the cache and memory state. The system recovers by undoing the logged changes. SafetyNet uses the standard solutions to the output commit problem (i.e., waiting to verify data before sending it to I/O devices) and the input commit problem (i.e., logging data received from I/O devices).

(4) Forward progress. We must ensure that, even in the worst-case mis-speculation scenario, the system continues to make forward progress. Mis-speculation must not fall victim to a pathological situation, whether unintentional or due to malicious software. As with detection, mechanisms for forward progress should not be overly complex. All three of our examples use similar forward progress mechanisms that are guaranteed to alter the timing of the execution after system recovery such that the race cannot recur.

In Section 3 and Section 4, we present three applications of speculation for simplicity, and we summarize them in Table 1. First, we simplify the design of a directory protocol by speculating that the adaptively routed interconnect provides point-to-point ordering of messages. Second, we simplify the design of a broadcast snooping cache coherence protocols by treating a rare corner case as a mis-speculation instead of explicitly designing for it. Third, we simplify the design of interconnection networks by removing the virtual channels used for deadlock avoidance. In all examples, detection and forward progress are easy to implement and thus speculation simplifies system design.

3 Simplifying Cache Coherence Protocols

In this section, we describe two ways to use speculation to simplify cache coherence protocols. Cache coherence protocols define the behaviors of the cache and memory controllers. Each controller is a finite state machine that has some number of states (per cache block) and handles some number of events that can happen to a block. Numerous controllers concurrently interact with each other with respect to many different blocks. While protocols are simple at a high level, they are much more complicated to design at a low level. Textbooks often abstract protocols into a handful of stable states (MOESI) and a handful of messages that nodes exchange [6]. In reality, though, protocols have numerous transient states, and messages race with each other in the interconnect and can arrive in many different orders.

Cache coherence protocols are notoriously difficult to design and verify. The state space explosion problem—an exponential function of the number of controllers, memory blocks, and block states—limits the viability of various formal verification methods [4], such as model checking and theorem proving. Testing is a valuable complement to formal verification techniques, and directed testing or randomized testing [3, 23] can uncover many bugs. Unfortunately, the complexity of protocols is often due to subtle race conditions, especially those that are infrequent and thus less likely to be uncovered by testing.

3.1 Simplifying a Directory Protocol

We now demonstrate how to simultaneously achieve (a) the design simplicity of a directory cache coherence protocol that relies on ordering in the interconnection network, and (b) the benefits of adaptive routing in the interconnect. We can simplify the design of a directory cache coherence protocol by relying upon the interconnect to provide *point-to-point ordering*, a property that guarantees that if a source sends two messages to a destination, then the messages arrive in the order in which they were sent. Point-to-point ordering eliminates certain potential races in the protocol, as we discuss later, and handling these races adds design and verification complexity.

Although point-to-point ordering can simplify the coherence protocol, most high-speed interconnect designs do not provide it. Interconnection networks can often achieve greater throughput and performance by using adaptive routing. Adaptively routed interconnects, such as that of the Alpha 21364 [16], allow two messages from switch S1 to switch S2 to take different paths. Adaptive routing can improve performance by distributing traffic more evenly across the interconnect and by enabling messages to be routed around localized congestion. Adaptive routing can also enhance availability by routing messages around faulty switches. Adaptive routing, however, does not provide point-to-point order in the interconnection network. In addition to adaptive routing, the use of reliable link-level retry mechanisms can preclude the provision of point-to-point ordering.

We illustrate an example of how adaptive routing can violate point-to-point order in Figure 1, in which a source node sends two messages to a destination node. The source sends message M2 after sending message M1, but M2 arrives first at the destination. The reversal in arrival order could be due, for example, to higher contention along the path taken by M1. With static routing, both messages would have followed the same path and arrived in order.

Specific Example. We explore a system with a MOSI directory cache coherence protocol and a two-dimensional (2D) torus interconnection network. There are four classes of messages in the protocol—Request, ForwardedRequest, Response, and FinalAck—and each class of messages travels on a logically separate interconnection network (i.e., virtual network). There are three types of Request mes-

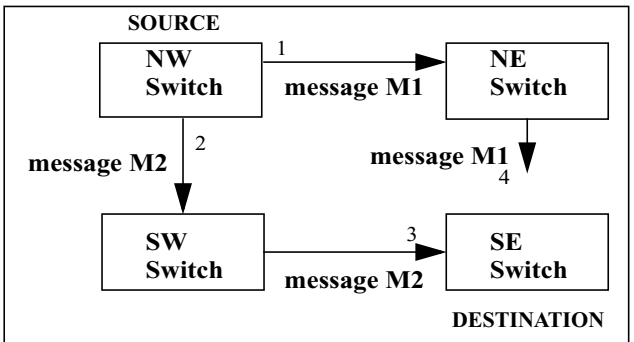

Figure 1. Violating point-to-point order with adaptive routing. The NW Switch sends Message M1 at time 1 and then message M2 at time 2 to the SE Switch. Message M2, however, arrives at time 3, which is before message M1 arrives at time 4.

sages that processors send to directories: RequestReadOnly, RequestReadWrite, and Writeback. There are four types of ForwardedRequest messages that directories send to processors: Forwarded-RequestReadOnly, Forwarded-RequestReadWrite, Invalidation, and Writeback-Ack. There are three types of Response messages that processors or directories send to requesting processors: Data, Ack, or Nack. Processors send messages to directories on the FinalAck virtual network to coordinate SafetyNet checkpoints, but we do not discuss them further here.

We can simplify directory protocol design by relying upon point-to-point order (per virtual network, but not across virtual networks) to avoid certain race cases. One common example of these races occurs when the owner of a block, processor P1, sends a Writeback to the directory and another processor, P2, sends a RequestReadWrite for the same block to the directory. If the RequestReadWrite arrives first, the directory then sends a Forwarded-RequestReadWrite and a Writeback-Ack to P1. If those messages, which travel on the same virtual network, arrive in the reverse order of that in which they were sent (i.e., the Writeback-Ack arrives before the Forwarded-RequestReadWrite), then P1 first sees the Writeback-Ack and downgrades to Invalid. Thus, it cannot handle the incoming Forwarded-RequestReadWrite. Designers of directory protocols can handle this race, but doing so adds additional states and transitions to the protocol and increases the complexity of protocol verification.

Instead of handling this race by adding extra states and transitions, we implemented a speculative system with an adaptively routed interconnection network and a directory cache coherence protocol that relies upon point-to-point ordering. The adaptive routing algorithm allows messages to choose among minimal distance paths based on outgoing queue lengths in each direction.[1]

(1) Infrequency of mis-speculation. The routing algorithm, while adaptive, is still unlikely to violate point-to-point ordering (we will show that it reorders <1% of messages). Moreover, even when it does violate ordering, few re-orderings impact correctness. Except in the example described above, re-ordering does not affect correctness, for several reasons. First, in this protocol, point-to-point ordering is only necessary on one virtual network (the ForwardedRequest virtual network). Second, ordering only matters for messages concerning the same block of memory. Third, even for messages concerning the same block, ordering only matters between certain message types. For example, the directory can send multiple Forwarded-RequestReadOnly messages to the owner of a block, but the order in which they arrive does not matter for correctness. In particular, the situation in which a Writeback-Ack races a Forwarded-RequestReadWrite is particularly rare, since a block just evicted at one node is unlikely to be actively wanted by another node.

(2) Detection. For this speculative system, a mis-speculation can only manifest itself as one particular invalid transition in a cache coherence controller, so cache controllers can detect all illegal message re-orderings. For our race case, a cache without a valid copy that receives a Forwarded-RequestReadWrite determines this situation to be a mis-speculation and triggers a system recovery. This mis-speculation cannot manifest itself in any other fashion and thus is easy to detect.

(3) Recovery. We use SafetyNet in all three of our speculatively simplified designs.

(4) Forward progress. To ensure forward progress, we alter the timing of the execution after recovery. We simply allow the interconnect to selectively disable adaptive routing, so that the system can always make forward progress. The choice of when to re-enable adaptive routing provides an adjustable knob for setting the worst-case lower bound on performance. At the conservative extreme, never re-enabling it would bound performance degradation, compared to static routing, to the cost of one mis-speculation.

3.2 Simplifying a Snooping Protocol

We now present an example of a protocol race in a broadcast snooping system that the designers (the authors!) did not initially consider. The designers overlooked this case until weeks later when randomized testing happened to uncover it (by crashing the simulator). Instead of forcing

1. While adaptive routing can break point-to-point order, it requires extra buffering to avoid deadlock. To isolate the issue of adaptive routing for purposes of this discussion, we simplistically avoid deadlock with full buffering. A realistic implementation would likely use a more clever solution, such as Duato's scheme for deadlock-free adaptive routing [8].

the designers to re-work the protocol and re-verify it, we explore the potential to reduce verification effort and speed the time to market by treating this edge case as a mis-speculation that triggers system recovery.

Specific Example. We developed a version of the snooping protocol system with support for speculation. The system treats a certain protocol corner case as a mis-speculation instead of handling it.

(1) Infrequency of mis-speculation. Mis-speculation occurs when a cache controller has a block in state Modified (or Owned) and then issues a Writeback for the block, transitioning to a transient state. In this transient state, a RequestForReadWrite arrives from another node, causing the cache controller to transition to a different transient state. Then, in this second transient state, the cache controller observes another RequestForReadWrite from another node. This sequence of events occurs exceedingly rarely, especially since it begins with a Writeback from the cache controller. Moreover, a block evicted by a Writeback is unlikely to be requested by two other nodes. Moreover, both nodes must request exclusive access to the block in the interval of time between when the cache controller issues its Writeback and then observes its own Writeback on the request network. While this scenario occurs rarely in practice, we still must handle it.

(2) Detection. The potential mis-speculation due to encountering this unspecified coherence transition can only manifest itself as one specific invalid transition and is thus easy to detect. In this particular example, a cache controller that observes another node's RequestForReadWrite while in the transient state described above detects the mis-speculation.

(3) Recovery. We use SafetyNet in all three of our speculatively simplified designs.

(4) Forward Progress. As with the prior example, we ensure forward progress by altering the timing of the execution after recovery. As a fail-safe mechanism, the system temporarily enters a "slow-start" mode, in which the system restricts the number of coherence transactions allowed to be outstanding (e.g., one). The corner case in the protocol can only occur when at least two transactions race in the system. The slow-start mode performance provides a worst-case lower bound on performance in the presence of mis-speculations. Moreover, before resorting to slow-start, the system could simply try to resume execution, perhaps in a slightly slower mode, in the likely hope that the race does not recur.

Along with the coherence protocol, the other primary source of complexity in a multiprocessor is the interconnection network, and we now discuss how to simplify it.

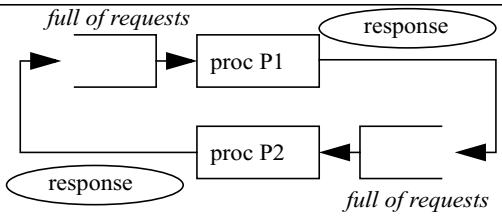

Figure 2. Endpoint deadlock

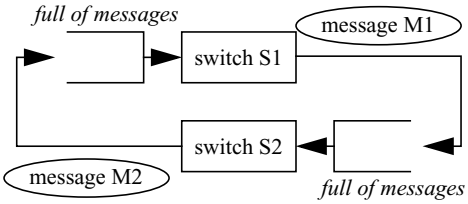

Figure 3. Switch deadlock

4 Simplifying the Interconnection Network

In this section, we discuss how to simplify deadlock avoidance in interconnection networks. Interconnects for multiprocessors are difficult to design, partly because of the difficulty of achieving high and robust performance while verifying that deadlock is impossible under all situations. There are two types of deadlock, which we refer to as endpoint deadlock and switch deadlock, based on where the deadlock can occur. We discuss both of them now, including the primary approach for avoiding them, before delving into a speculative design that is simpler.

Endpoint deadlock. Endpoint deadlock can occur when cross-coupled requests depend on each other, as shown in Figure 2. For example, deadlock can occur if (a) processor P1 sends a request for block A to P2 followed by a response for block B, (b) P2 does the opposite (request for B followed by response for A), (c) the incoming queues for both processors are full of requests, and (d) neither processor can ingest its incoming request until it ingests its incoming response, and they process incoming buffers in order. This type of deadlock depends on the coherence protocol, but it is independent of interconnection network topology and routing.

Switch deadlock. Switch deadlock in the interconnection network can arise due to the combination of cross-coupled messages and insufficient buffering for in-flight messages. Consider the simple example illustrated in Figure 3. In this example, switch S1 wants to send message M1 to switch S2, and S2 wants to send M2 to S1. However, the buffer from S1 to S2 and the buffer from S2 to S1 are both full and unable to accept new messages. Moreover, neither switch will process its incoming queue until it can send its

outgoing message. Thus, if switches process incoming message buffers in FIFO order, the system now deadlocks, since neither M1 nor M2 can make progress. This type of deadlock depends on the topology of the interconnect and the routing policy, but it does not depend on the cache coherence protocol.

To avoid both types of deadlock, interconnects can either use worst-case buffering or some scheme to break the cyclic dependences that can lead to deadlock. Using worst-case buffering at each endpoint and switch is the simplest solution, but it is generally not viable since the worst case can be far worse than the common case.

To avoid the costs of worst-case buffering, many interconnection networks use schemes that break the cyclic dependences that can lead to deadlock [10]. To avoid endpoint deadlock, we can use virtual networks for the different classes of messages. A virtual network is just one or more incoming buffers reserved by the switches and endpoints for a particular class of messages. If we have a different virtual network for each class of messages (e.g., request and response), then the incoming queue can never fill up with just requests, since we have reserved space for responses. To avoid switch deadlock, we can use virtual channel flow control [7]. A virtual channel is just one or more buffers per unidirectional physical link[2] that is reserved by the switch for messages of a particular priority. In our simple example, if M1 was on virtual channel 1 and M2 was on virtual channel 2, then deadlock would not occur. The interaction of virtual channels and virtual networks is multiplicative; if we need *N* virtual networks to avoid endpoint deadlock and *C* virtual channels to avoid switch deadlock, then we need *NxC* virtual channels total (i.e., *C* per virtual network). For a 2D bidirectional torus and our directory cache coherence protocol, we require two virtual channels per virtual network (for static routing) and four virtual networks (i.e., 4*2=8 virtual channels total). To provide deadlock freedom with adaptive routing requires at least one additional virtual channel [9].

Virtual channel/network flow control minimizes deadlock-free buffering requirements and it is well-understood, but it adds complexity and requires additional verification effort. The SGI Origin directory protocol [13] avoids this complexity by using only two virtual networks instead of the three that would have ensured deadlock avoidance. Instead, the Origin relies on a higher level mechanism to negatively acknowledge (nack) its way out of the deadlocks that occur due to this limitation, even though nacking increases protocol complexity and could introduce livelock problems. At the other extreme, the Alpha 21364 interconnect uses nineteen virtual channels (six virtual networks times three virtual channels each, plus an extra channel for special messages) [16], demonstrating that complicated flow control is implementable.

As an alternative to virtual channel/network flow control, interconnect designers have used deflection routing (a.k.a. hot potato routing) to avoid deadlock. Deflection routing avoids deadlock without using any buffering, but it can suffer from potential livelock problems.

Specific Example. To demonstrate the viability of easing interconnection network design, we implemented a 2D torus interconnect with less than worst-case buffering and no virtual channel/network support. We use the same system model as the directory protocol example in Section 3.1. In the case that the system detects deadlock, it recovers and resumes execution.

(1) Infrequency of mis-speculation. Adaptively routed networks are generally designed to avoid potential deadlock conditions. This is because both deadlock avoidance (e.g., using "escape channels" [8]) and deadlock recovery (e.g., as proposed here) negatively impact performance. Designers can size buffers to reduce the probability of potential deadlocks. Some networks also use source throttling to further reduce this probability [21]. Using recovery to resolve deadlock changes the problem from a correctness issue into a performance issue.

(2) Detection. Detection of this form of mis-speculation is straightforward, since the requestor can detect all deadlocks by a time-out.[3] If a message gets stuck in the interconnect, the coherence transaction to which it belongs will not complete, and the requestor of the transaction will timeout and trigger a system recovery. We choose time-out latency to be long enough to mitigate false positives while short enough to not slow down SafetyNet commitment of checkpoints.[4] Since there is little gain in having a timeout latency shorter than necessary for SafetyNet, a processor times out on its request after three checkpoint intervals.

(3) Recovery. We use SafetyNet in all three of our speculatively simplified designs.

(4) Forward Progress. As with the prior two examples, we ensure forward progress by altering the timing of the execution after recovery. As a fail-safe mechanism, we enter a "slow-start" mode like that in Section 3.2, in which the sys-

2. When we refer to virtual channel requirements, they are the requirements *per unidirectional physical link*.

3. A timeout mechanism would also detect livelock if we were to use speculation to support deflection routing on a topology for which deflection routing does not provably avoid livelock.

4. SafetyNet cannot commit an old checkpoint until it is sure that execution prior to that checkpoint was mis-speculation-free. Thus, it might have to wait as long as the timeout latency to either commit a checkpoint or trigger a system recovery.

Table 2. Target System Parameters

Memory System	L1 Cache (I and D)	128 KB, 4-way set associative
	L2 Cache	4 MB, 4-way set-associative
	Memory	2 GB, 64 byte blocks
	Miss From Memory	180 ns (uncontended, 2-hop)
	Interconnection Networks	link bandwidth = 400MB/sec to 3.2 GB/sec
SafetyNet	Checkpoint Log Buffer	512 kbytes total, 72 byte entries
	Checkpoint Interval	100,000 cycles (directory), 3,000 requests (snooping)
	Register Checkpointing Latency	100 cycles

tem restricts the number of coherence transactions allowed to be outstanding. As long as we provide enough buffering to satisfy the reduced number of transactions, slow-start provably avoids livelock and provides a worst-case lower bound on performance. We can raise this bound by providing more buffering to support a slow-start mode that allows more concurrent transactions. Moreover, before resorting to slow-start, the system could simply try to resume execution, possibly in a slower mode, in the hope that deadlock will not occur again. If deadlock does recur, the system can then fall back on slow-start. Slow-start adds some complexity back into the system, but it is less than that of virtual channel flow control.

5 Evaluation

In this section, we evaluate the applications of speculation for simplicity that we have developed in Sections 3 and 4. Our first goal is to determine the mis-speculation frequency at which our system performance begins to degrade. Our second goal is to determine if our speculative systems incur mis-speculations infrequently enough. We start by describing our system model, methodology [1], and workloads, and then we discuss our results.

5.1 System Model and Simulation Methodology

Our target system is a 16-node shared-memory multiprocessor. Each node consists of a processor, two levels of cache, some portion of the shared memory and directory, and a network interface. The processor architecture is SPARC v9, and the system runs Solaris 8.

We evaluate our target system with the Simics full-system, multiprocessor, functional simulator, and we extend Simics with a memory hierarchy simulator to compute execution times. Simics is a system-level architectural simulator developed by Virtutech AB [15]. We use Simics/sun4u, which simulates Sun Microsystems's SPARC V9 platform architecture (e.g., Sun E6000s) in sufficient detail to boot unmodified Solaris 8. Simics is a functional simulator only, and it assumes that each instruction takes one cycle to execute (although I/O may take longer), but it provides an interface to support detailed memory hierarchy simulation.

Processor. We model a processor core that, given a perfect memory system, would execute four billion instructions per second and generate blocking requests to the cache hierarchy and beyond. We use this simple processor model to enable tractable simulation times for full-system simulation of commercial workloads. While an out-of-order processor model might affect the absolute values of the results, mostly due to being able to maintain more outstanding memory requests, it would not qualitatively change them. If having additional outstanding requests leads to more exercising of certain races or corner cases, we could violate the first necessary feature of speculation for simplicity (i.e., infrequency of mis-speculation). Then we would have to re-consider this particular speculation but not speculation for simplicity in general.

Memory System. We have implemented a memory hierarchy simulator that supports our coherence protocols and SafetyNet. The simulator captures all state transitions (including transient states) of our coherence protocols in the cache and memory controllers. We model the interconnection network topologies and the contention within them. In Table 2, we present the design parameters of our target memory systems.

SafetyNet. For our system recovery mechanism, we use SafetyNet [19]. We list SafetyNet system parameters in Table 2, and SafetyNet performance overhead (in error-free execution) is minimal for these design parameters, as was demonstrated by Sorin et al. [19]. The checkpoint interval differs between the directory and snooping systems, due to the different logical time basis used for creating consistent checkpoints. Recovery time varies somewhat, depending on how much work the system loses between the recovery point and when it detects the mis-speculation. We stress-tested the recovery mechanism by periodically triggering recoveries, and we show these results in Section 5.3.

5.2 Workloads

Commercial applications represent an important workload for shared memory multiprocessors. As such, we evaluate our speculative design with four commercial applications and one scientific application, described

Table 3. Workloads: Wisconsin Commercial Workload Suite and a Splash-2 scientific workload

OLTP: Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use a 1 GB 10-warehouse database stored on five raw disks and an additional dedicated database log disk. There are 8 simulated users per processor. We warm up for 10,000 transactions, and we run for 500 transactions.
Java Server: SPECjbb2000 is a server-side java benchmark that models a 3-tier system with driver threads. We used Sun's HotSpot 1.4.0 Server JVM. Our experiments use 24 threads and 24 warehouses (~500 MB of data). We warm up for 100,000 transactions, and we run for 10,000 transactions.
Static Web Server: We use Apache 1.3.19 (`www.apache.org`) for SPARC/Solaris 8, configured to use pthread locks and minimal logging as the web server. We use SURGE to generate web requests. We use a repository of 2,000 files (totalling ~50 MB). There are 10 simulated users per processor. We warm up for ~80,000 requests and run for 5000 requests.
Dynamic Web Server: Slashcode is based on a dynamic web message posting system used by `slashdot.com`. We use Slashcode 2.0, Apache 1.3.20, and Apache's `mod_perl` 1.25 module for the web server. MySQL 3.23.39 is the database engine. The database is a snapshot of `slashcode.com`, and it contains ~3,000 messages. A multithreaded driver simulates browsing and posting behavior for 3 users per processor. We warm up for 240 transactions and run for 50 transactions.
Scientific Application: We use *barnes-hut* from the SPLASH-2 suite [22], with the 16K body input set. We measure from the start of the parallel phase to avoid measuring thread forking.

briefly in Table 3 and in more detail by Alameldeen et al. [1]. To address the variability in runtimes for commercial workloads, we simulate each design point multiple times with small, pseudo-random perturbations of memory latencies to cause alternative scheduling paths [1]. Error bars in results represent one standard deviation in each direction.

5.3 Results

We now present the results of our evaluations of each of our three speculative designs. We first explore the impact of mis-speculation, in general, by stress-testing the system's ability to recover from a range of mis-speculation rates. While mis-speculation occurs infrequently in our speculative systems (shown next), we want to determine the mis-speculation rate at which the latency of recoveries can impact performance. To isolate the impact of increasing the frequency of recoveries, we implement a system without speculation and inject periodic recoveries. The results, shown in Figure 4, reveal that recovery is sufficiently short that the performance cost of recovering even ten times per second is negligible.[5]

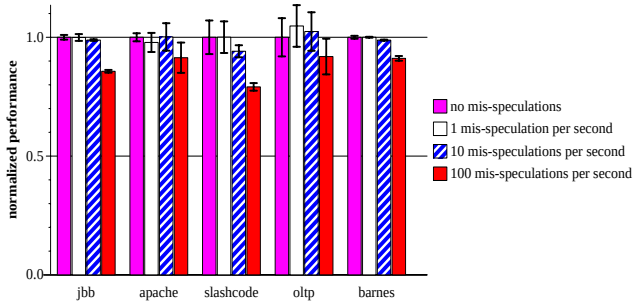

Figure 4. Performance vs. Mis-speculation Rate

Speculatively Simplified Directory Protocol Results. We evaluated the performance of the speculatively simplified directory protocol to determine if mis-speculations are sufficiently infrequent to make speculation viable. Re-ordering and mis-speculation rates are a function of available bandwidth, since increasing the bandwidth provides fewer opportunities for adaptive routing. The primary result is that virtually no reorderings occur in our system, even for link bandwidths as low as 400 MBytes/sec.

Adaptive routing incurs few recoveries for two reasons. First, re-orderings are rare, even for link bandwidths of 400 MBytes/sec. With mean link utilizations between 13-35% (for static routing), there is little opportunity for adaptive routing to re-order messages to avoid congestion. Second, when re-orderings do occur, the vast majority of them do not affect correctness. While adaptive routing re-ordered 0.1-0.2% of all messages on the ForwardedRequest virtual network, we observed only a handful of recoveries in all simulations. On other virtual networks, adaptive routing re-ordered as many as 0.8% of messages, but these re-orderings cannot violate correctness in our protocol. It also re-ordered messages across virtual networks (e.g., a request arrived after a response despite being sent first), but this re-ordering does not matter in our protocol.

While it might thus appear that adaptive routing is not worthwhile, it can still help performance during periods of higher congestion. Although *mean* link utilization is low, *instantaneous* utilization is sometimes much greater. In these instances, adaptive routing can route more messages around congestion. In Figure 5, for an interconnect with link bandwidth of 400 MBytes/sec, we compare the relative performances of systems with static and adaptive routing, and we normalize the results to the performance of static routing. We observe that adaptive routing achieves a significant speedup for our workloads because of better instantaneous link utilization and the infrequency of recoveries.

5. Mis-speculation does not *improve* performance for OLTP. The error bars show that, despite the greater mean performance value with mis-speculations, the performance is not statistically significantly better.

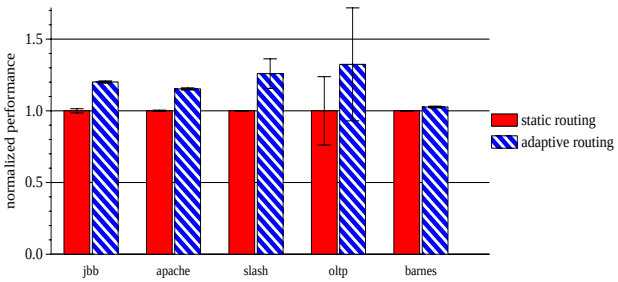

Figure 5. Relative performances of static and adaptive routing (400 Mbytes/sec link bandwidth)

In summary, designers can simplify a directory protocol by speculatively relying on point-to-point ordering despite the use of adaptive routing. Moreover, the use of adaptive routing may allow for cheaper links and fewer pins, while achieving the same bandwidth as a statically routed yet more costly interconnect.

Speculatively Simplified Snooping Protocol Results. We tested the speculatively simplified snooping coherence protocol on our set of commercial workloads, and all of them ran to completion without needing to recover even once from reaching the edge case. Thus, performance of the protocol mirrors, for these workloads, that of the fully designed protocol. While this observed lack of recoveries obviously does not guarantee that the speculative protocol will never have to recover, it does suggest the infrequency of recoveries due to encountering this corner case in the cache coherence protocol.

We conclude from this experiment that a system can tolerate a corner case in the cache coherence protocol with speculation. Even if recoveries due to mis-speculations slightly degrade performance, the reduced time for design and verification provides a benefit that is likely to more than offset the runtime cost of recoveries.

Simplified Interconnection Network Results. To determine the performance impact of recoveries, we compare the performance of this system against a system with the same protocol running on an interconnection network with worst-case buffering (given no virtual channels/networks). Results (not plotted) show steady performance for systems with buffer sizes at and above 16 but a sharp dropoff in performance for systems with buffers of size 8. Deadlocks do not occur in any of our workloads until we reduce buffer sizing from 16 to 8. Although our 16-processor system can only have a total of 16 *requests* outstanding, deadlock with buffers of size 16 is still possible because (a) each processor can also have a writeback outstanding, (b) each request can be forwarded from the directory to multiple destinations, and (c) all message types share the same buffers.

We conclude from this experiment that we can simplify interconnection network design with speculation. We can remove virtual channel deadlock avoidance and speculatively assume that deadlock will not occur. When it does, the system can recover and still make forward progress.

Summary of Results. These experimental results show that speculation is a viable technique for simplifying system design. In general, a speculative system can maintain its performance even when ten recoveries per second occur, and our speculative systems incur recoveries less frequently than that.

6 Related Work in Simplifying System Design

There exists some prior research in simplifying system design by allocating resources towards common-case scenarios. We illustrate four notable examples of this approach.

The first example is the use of software for solving complex processor hardware problems. For example, processors have trapped to software for floating point arithmetic, such as the Intel 80386 without the 80387 floating point coprocessor. No recovery is necessary for these traps to software. Also, numerous architectures give the user the option of trapping to software for IEEE standard denormalized floating point arithmetic, including SPARC v9 [20] and Intel IA-64 [12]. Localized processor recovery may be necessary in these cases to support precise interrupts.

Second, prior research has explored the use of fewer virtual networks than strictly necessary in all cases. The SGI Origin [13] can detect deadlock and then fall back on a higher-level mechanism, specified only vaguely in the public literature, to undo the deadlock and enable forward progress. Similarly, more general progressive deadlock recovery schemes have been proposed for endpoint deadlock [18]. The MIT Alewife uses software to recover from interconnect deadlock due to insufficient resources in rare situations [14].

Third, some processors have relied on the inherent recoverability of dynamically scheduled processors to recover from corner cases. For example, the Pentium Pro flushes its pipeline when it manipulates certain control registers [11]. The Intel Pentium 4 recovers from deadlocks due to corner cases in scheduling the processor core [5]. After recovery, the core will not deadlock, since the recovery (and flush) changes the microarchitectural state enough to avoid the same corner case that originally caused the deadlock.

Lastly, DIVA [2] uses a simple checker processor to dynamically verify a complex and possibly faulty processor and ensure correctness even in the case of design errors.

DIVA is the closest work to speculation for simplicity, but DIVA is limited to only the processor core.

7 Conclusions

In this paper, we have discussed how to use speculation to simplify multiprocessor system design. Speculation allows the designer to allocate design and verification effort towards the common case scenarios while falling back on system-wide recovery for situations that are rare and complicated. The three applications of speculation for simplicity that we have presented have demonstrated the potential to simplify system design.

Acknowledgments

We would like to thank José Duato, Alvy Lebeck, Amir Roth, and the Wisconsin Multifacet Group for their helpful discussions of this work.

References

[1] A. R. Alameldeen et al. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[3] R. M. Bentley. Validating the Pentium 4 Microprocessor. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 493–498, July 2001.

[4] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.

[5] B. Colwell. Personal Communication, June 2002.

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

[7] W. J. Dally. Virtual Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, Mar. 1992.

[8] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, Dec. 1993.

[9] J. Duato and T. M. Pinkston. A General Theory for Deadlock-Free Adaptive Routing Using a Mixed Set of Resources. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1219–1235, Dec. 2001.

[10] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks*. IEEE Computer Society Press, 1997.

[11] Intel Corporation. *Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual*. Intel Corporation, 1995.

[12] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture, Revision 1.1*, July 2000.

[13] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[14] K. Mackenzie et al. Exploiting Two-Case Delivery for Fast Protected Messaging. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, Feb. 1998.

[15] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[16] S. Mukherjee et al. The Alpha 21364 Network Architecture. In *Proceedings of 9th Hot Interconnects Symposium*, Aug. 2001.

[17] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, May 2002.

[18] Y. H. Song and T. M. Pinkston. A Progressive Approach to Handling Message-Dependent Deadlock in Parallel Computer Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(1):1–17, Jan. 2003.

[19] D. J. Sorin, M. M. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.

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

[21] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee. Self-Tuned Congestion Control for Multiprocessor Networks. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

[22] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, June 1995.

[23] D. Wood, G. Gibson, and R. Katz. Verifying a Multiprocessor Cache Controller Using Random Test Generation. *IEEE Design and Test of Computers*, pages 13–25, Aug. 1990.