Memory Systems for Parallel Programming

by

Bradley Eric Richards

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin — Madison

1996

Abstract

Distributed Shared-Memory (*DSM*) computers, which partition physical memory among a collection of workstation-like computing nodes, are emerging as the way to implement parallel computers, as they promise scalability and high performance. Shared-memory DSM machines use a coherence protocol to manage the replication of data and to ensure that a parallel program sees a consistent view of memory.

Applications have very different patterns of communication and no single, general-purpose protocol suits all programs. This has prompted interest in systems in which a protocol is implemented in flexible software instead of being fixed in hardware. DSM machines with software-implemented coherence protocols provide opportunities for a variety of more complex and application-specific protocols and *allow for protocols that do not just ensure consistent memory, but also provide new functionality and semantics.*

Parallel programming has long faced a tension between the goals of high performance and ease of use. Languages and tools can make parallel computers easier to use, but concerns about their efficiency have limited their usage. This thesis demonstrates that some high-level languages and tools can be implemented more efficiently by taking advantage of the cache coherence protocols that underly software DSM machines, thereby improving both performance *and* ease of use.

This thesis describes a family of custom protocols that efficiently implement a large-grain data-parallel language C**. On programs for which static analysis is imprecise, these Loosely Coherent Memory (LCM) protocols improve performance from a few percent up to a factor of 3, and reduce memory overheads from a factor of 2 to a factor of 5 over a compiler-copying scheme. LCM is also improves performance in C-code programs by up to a factor of 3.

This thesis also presents custom cache-coherence protocols that perform on-the-fly detection of actual data races for programs with barrier synchronization. Overheads in execution time for the race-detection protocols were shown to range from zero to less than a factor of three — a significant improvement over comparable approaches — and race-detection protocols found actual program errors in two applications.

Acknowledgments

It is a pleasure to finally acknowledge those who have helped me achieve this goal. At the top of the list are my parents and grandparents. Without their financial and emotional support over the years, I would not be where I am today. I also thank Jim Larus, my advisor, for his support and encouragement, and for helping to sharpen my research and presentation skills. Thanks also to Bart Miller and Charles Fischer, for taking the time to read this thesis and providing valuable feedback.

That these past six years of graduate school have been so enjoyable is entirely due to the people with whom I have worked and played. The members of the Wisconsin Wind Tunnel group were a source of constant support and stimulation. Alvy Lebeck, Babak Falsafi, and Steve Reinhardt deserve special mention for always making time to answer my questions. I will particularly miss my interactions with Satish Chandra and Guhan Viswanathan — they were always enlightening and enjoyable.

Several generations of graduate students helped to brighten my days in Madison. Cheryl Thompson, Dan Ross, and Tim Morrison made me feel welcome from the moment I arrived. When they moved on, Kurt Brown, Bill Roth, and Mary Tork Roth looked after me. These last few years I have had the pleasure of sharing the company of Mark Craven, Susan Goral, Susan Hert, Tia Newhall, and Martha Townshend. Susan Hert deserves special recognition for being brave enough to share a house with me for the last three years, and for being such a good friend. Kelly Cherry, Pedar Foss, Pete Machalek, Jill Smook, and the rest of the college gang kept in touch, and provided valuable distractions from my studies. Last but not least, my thanks to Holly for her support during these last few difficult months. Having you there made all the difference.

BRADLEY ERIC RICHARDS

University of Wisconsin — Madison

August 1996

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The primary motivation underlying parallel computing is simple: Users can obtain higher performance by dividing a computation across a set of processors and running portions of it concurrently. Unfortunately, as many have discovered, programming parallel computers can be much more difficult than programming sequential computers. The task is easier if a parallel system supports a shared address space, since this abstraction allows processors to share a common pool of memory and frees a programmer from concerns about the correctness of data layout and movement. Distributed Shared-Memory (*DSM*) computers, which partition the physical memory among a collection of workstation-like computing nodes, are emerging as a popular way to implement parallel computers because they promise scalability and high performance.

Shared-memory DSM machines require a coherence protocol to manage the replication of data and to ensure that a parallel program sees a consistent view of memory [3, 19, 32, 42, 64]. In general, coherence protocols allow at most a single processor to modify a shared location, either invalidating outstanding copies or updating copies with the new value. A protocol determines, to a large extent, the performance of a shared-memory program since communication occurs through loads and stores to shared data.

But, applications have very different patterns of communication, and no single, general-purpose protocol has proven well suited to all programs. This has prompted interest in systems that enable users to select from a set of coherence protocols [13, 18] and, more recently, in systems in which a protocol is implemented in flexible software instead of being forever encoded in hardware [36, 54]. Experiments have shown that the performance penalties for implementing coherence actions in software, instead of hardware, are relatively small (especially if common operations are accelerated by hardware [36, 55]), and that tailoring protocols to the needs of applications can result in tremendous performance increases [28].

Parallel computers of the future will likely be DSM systems with software-implemented coherence protocols. This hardware provides opportunities for a large variety of more complex and application-specific protocols and *allows for protocols that do not just ensure consistent memory, but also provide new functionality and semantics*. Tasks ranging from program monitoring (i.e. bounds checking, profiling, performance monitoring) to language implementation (i.e. garbage collection, reductions) can benefit from access to protocol-level information and performance.

Parallel programming has long faced a tension between the goals of high performance and ease of use. Languages and tools can make parallel computers easier to use, but concerns about their efficiency have limited their usage. This thesis demonstrates that some high-level languages and tools can be implemented more efficiently by taking advantage of the cache coherence protocols that underly software DSM machines, thereby improving both performance *and* ease of use. The following sections describe in more detail the contributions this thesis makes towards efficiently implementing a new parallel programming language and detecting data races through the use of custom coherence protocols.

1.1 Loosely Coherent Memory

Recently, there has been considerable interest in higher-level parallel languages, such as HPF [31], in which a compiler handles the details of mapping from an abstract parallel model to a particular machine. The success of this, and other, parallel languages depends on efficient implementations.

This thesis describes a new approach to implementing languages on parallel machines in which a cache coherence protocol called Loosely Coherent Memory (*LCM*) plays a crucial role. The compiler and LCM collaborate to efficiently implement the semantics of a high-level parallel language by exploiting user-level control of a processor's address space to detect and handle language semantic exceptions without slowing down execution of correct memory references. This collaborative effort is similar to sequential language implementations, such as Lisp and ML, that exploit memory systems to perform heap bounds checking and concurrent garbage collection [7]. In both systems, operations proceed under the assumption they will not fail. A processor's memory system catches unexpected situations, which are handled out of the program's normal line of code.

As a proof of concept, I show how LCM can be used to efficiently implement a large-grain data-parallel language called C** [39], whose semantics are otherwise difficult to implement. In C**, a parallel function is applied *simultaneously* and *instantaneously* to each element in an aggregate. This language semantics makes it appear as if each function invocation is the only one executing, which frees the programmer from considerations of conflicting side effects, and simplifies reasoning about program behavior. In a C** program running under LCM, memory references in parallel functions that could possibly interfere with other function invocations are specially marked. The LCM memory system maintains per-processor copies of modified cache blocks, which the memory system merges at the end of a parallel function invocation. Programs supported by LCM are shown to run up to a factor of three faster than those relying on conservative compiler-generated code to ensure correct C** semantics.

LCM works well with C**, but is not limited to the implementation of C** programs. LCM can also be used to optimize the performance of shared memory programs written in other languages by reducing the overheads incurred by both false and true sharing, and by streamlining communication. LCM is shown to increase performance by nearly a factor of five for some shared memory programs written in C and hand-annotated with the appropriate LCM directives.

1.2 Data Race Detection

A race condition exists in a shared-memory parallel program when accesses to shared memory are not properly synchronized. These unsynchronized references potentially indicate a programming error, since the order in which the references are completed, and so possibly the final value of the shared memory location, is undetermined. It can be difficult to debug programs containing data races, since the perturbation caused by the monitoring or debugging tools can cause the races to disappear. Thus, there has been much interest in techniques for finding and detailing races.

Static techniques can find *potential* races at compile time, but the static analysis and assumptions about code that could potentially execute concurrently must both be conservative, leading to reports of races that may not be possible. The programmer must then decide which reported races are spurious and which are truly problematic. Off-line analysis of trace information collected during an execution can be used to detect both races that actually occurred, and those that *could* have occurred given the synchronization contained in the program [48], but they require that large traces be generated and stored. On-the-fly checking can be used to detect races during execution, but the monitored programs have been shown to be up to a factor of six slower than the unmonitored [25].

Since cache coherence protocols manage accesses to shared data, they are aware of an application's memory references. This access information can be used, either directly or with augmentation, to detect actual data races during execution. Since the access information is already maintained by the coherence protocol, protocol-based approaches hold the promise of being able to detect races with lower overheads than existing techniques. This thesis presents results for a family of race-detection protocols showing that accurate, useful race detection can be performed with performance penalties of less than a factor of three.

1.3 Contributions

This thesis observes that cache coherence protocols can be used to help implement parallel programming language semantics, and identifies the points in coherence protocols at which compilers can control memory-system policies. It provides implementations of a set of custom protocols that support the parallel programming language C**, and formally verifies that they work correctly. Factors influencing protocol performance are investigated, and it is demonstrated that they can implement the semantics of C** more efficiently than an approach based on static compiler analysis. The protocols are also shown to be useful for supporting programs in languages other than C**.

This thesis presents implementations of a range of custom protocols that detect actual data races, formally verifies that they work correctly, and demonstrates that they can find data races in real applications. Tradeoffs between performance penalties and the accuracy of race detection are investigated, and it is shown that accurate detection of actual races can be performed with overheads ranging from none to a factor of less than three.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 contains background information on distributed shared-memory machines and cache coherence protocols. Chapter 3 describes the LCM protocol in detail and presents performance results for a set of C** programs supported by LCM. Chapter 4 shows how LCM can be used to improve the performance of shared-memory programs independent of C**. The race detection protocols are discussed in Chapter 5. Chapter 6 summarizes the thesis and gives directions for future work. Protocol finite-state machine diagrams are given in Appendices A and B.

Chapter 2

Background

Subsequent chapters describe how custom cache coherence protocols can help efficiently implement parallel languages and tools on distributed shared-memory computers. This chapter lays the groundwork by providing an overview of both distributed shared-memory systems and coherence protocols.

2.1 Distributed Shared Memory Systems

Distributed shared-memory systems present users with the illusion of a global, shared pool of memory even though memory is physically divided across a set of distinct processing nodes. They accomplish this by transparently moving copies of data from processor to processor in response to shared-memory accesses. DSM systems can be built using two basic mechanisms. The first mechanism, *access control*, allows the system to control access to memory by permitting read and write accesses only for valid, cached data. Reading or writing an invalid location or writing a valid but read-only location must cause an *access fault* and invoke the coherence protocol. The second mechanism, *communication*, enables a system to transfer control information and data among processors.

Access control can be performed at various granularities. *Page-based* systems [10, 12, 14, 17, 34, 44] typically use operating-system page-protection schemes to implement access control, and therefore enforce coherence at the page granularity. This approach can be used to implement shared memory on loosely-coupled systems where no hardware support for shared memory exists. But, the large coherence granularity can cause increased contention if programs share data at granularities finer than a page, as processors compete to gain access to the page-sized regions of memory. There can also be large overheads associated with transferring page-sized regions of memory. At the other extreme, Cache-Coherent Nonuniform Memory Access (*CCNUMA*) systems [2, 20, 43, 55] implement access control at the granularity of a cache block, reducing the contention over each block of memory.

Both access-control mechanisms and the coherence protocols that they invoke can be implemented in hardware or software. Traditionally, page-based DSMs have taken an all-software approach while CCNUMA designs have implemented both mechanisms and protocols in hardware. These lines have recently begun to blur. Fine-grained access control has been successfully implemented in software [58], and a number of CCNUMA machines have moved their coherence protocols to software as well [36, 45, 54]. Software protocols offer the possibility of tailoring protocols to the needs of applications, and makes the work in this thesis both possible and relevant.

2.2 Coherence Protocols

DSM systems implement a shared memory by transparently moving copies of data from processor to processor in response to shared-memory accesses. A coherence protocol ensures that a parallel program sees a consistent view of memory by managing the replication and movement of this data. In general, a protocol ensures some form of consistency either by invalidating outstanding copies when a processor writes to a memory location, or by updating these copies with the new value.

A protocol comes into play at an access fault. It must satisfy the faulting access by bringing data to the memory of the faulting processor. In many protocols, each block of shared data has a home node that coordinates accesses to the block. The accessing processor sends a request to the home of the referenced block, which performs bookkeeping duties and returns the data. Once a processor obtains the data, it caches a copy, which can be subsequently accessed until it is invalidated. Many protocols, for example the Stache protocol [55], enforce coherence by permitting only a single writer (or multiple readers) to a block. When a home node receives a request for a writable copy of a block, it invalidates the outstanding read-only copies before returning the writable block. The memory reference and the request and invalidation messages are protocol events, which cause transitions in a protocol state machine.

Both the home node and caching processors record state for a block. At a protocol event for block B, the protocol consults the state of B to determine an action. Actions may send messages to other processors, await their replies,

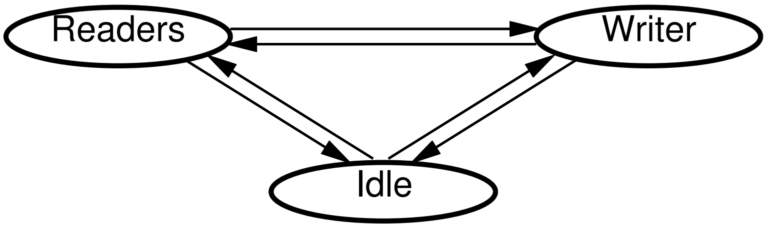


Figure 2.1: Simple home-side protocol finite-state machine

update protocol-specific information, and change access permissions. The exact states, transitions, and actions depend on the coherence algorithm. Many coherence schemes have been proposed [3, 11, 19, 32, 42], but none works well for all applications and sharing patterns.

Conceptually, a simple invalidation protocol like Stache requires only the three block states shown in Figure 2.1. A block is either *Idle*, in which case there are no remote cached copies, or there are one or more *Readers*, or a single *Writer*. Although the transitions appear as *atomic* state changes in response to protocol events, in reality they cannot be atomic. To avoid deadlock in a real system, protocol handlers must run to completion and terminate. In practice, this requires introducing *intermediate states* into the protocol finite-state machine.

Consider the transition from Writer to Readers in Figure 2.1, which responds to a read request by a processor. This transition can complete only when the block's previous owner relinquishes it. Conceptually, the action for this transition sends an invalidation message and awaits an acknowledgement. But handlers cannot wait on an asynchronous event, such as a message arrival. Hence, after sending the invalidate message, the handler must change to an intermediate state and

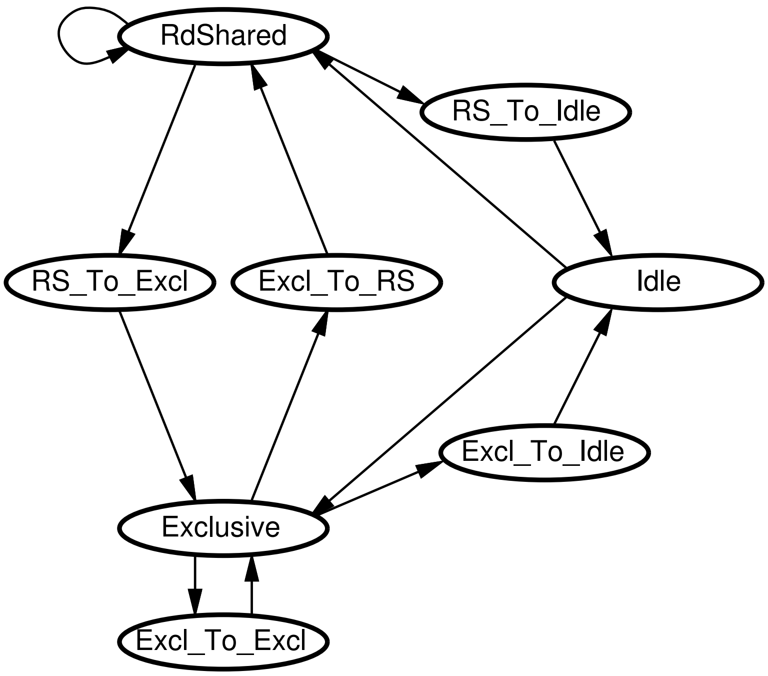


Figure 2.2: Finite-state machine with intermediate states

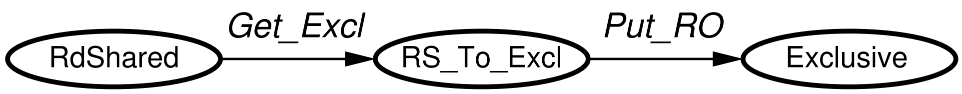
terminate. Figure 2.2 shows the state machine after introducing the necessary intermediate states. When the invalidation acknowledgement subsequently arrives, the transition completes by changing from the `Excl_To_RS` intermediate state to `RdShared`. Other states also require intermediate states for their transitions.

Intermediate states complicate programming because they make transitions non-atomic. While in an intermediate state, the protocol may receive many messages other than the expected reply message. For example, the state `Excl_To_RS` waits for an invalidation acknowledgement message. Before that message arrives, another processor may request permission to read or write the same block. The protocol designer must consider these possibilities and provide the `Excl_To_RS` state with suitable actions.

In general, the complexity of writing a correct protocol grows with the number of required states and the interactions between states and messages. Message reordering in the network further adds to the complexity, because messages may arrive in an unexpected order. For example, a read request from a processor that already has a readable copy cannot be ignored or treated as an error. The processor may have returned its copy and subsequently requested a readable copy. If messages can pass each other in the network, the read request must be retained and processed after the first copy has been returned.

2.2.1 Writing Protocols with Teapot

Teapot [22] is an environment for designing, verifying, and implementing cache coherence protocols. It simplifies the task in two significant ways. First, it provides



Get_Excl Handler:

```
Begin
   Foreach RO copy
      Send invalidation
   Move to RS_To_Excl
End
```

Put_RO Handler:

```
Begin
   Update bookkeeping
   If last RO copy
      Send Excl copy
      Move to Exclusive
End
```

Figure 2.3: Pseudocode handlers for `RdShared` to `Exclusive` transition

language support for organizing and writing protocol handlers. Second, it can help ensure that protocols work correctly by generating input for a formal verification tool called Murϕ [24]. Experience with Teapot has shown it to reduce protocol implementation and debugging time by an order of magnitude.

Language Support

The Teapot protocol specification language provides a construct, `Suspend`, that allows handlers to be written as though they can wait on asynchronous events. The handler is later compiled into atomically-executable pieces. To illustrate its use, consider the protocol transition from `RdShared` to `Exclusive` in response to a request for a writable copy. Before a writable copy can be given to a remote processor, all outstanding read-only copies must be invalidated. Figure 2.3 shows the pseudocode handlers required to implement this transition.

When the `Get_Excl` request arrives in state `RdShared`, the first handler is

Get_Excl Handler:

```
Begin
   Foreach RO copy
      Send invalidation
   Suspend(RS_To_Excl)
   Send Excl Copy
   Move to Exclusive
End
```

Put_RO Handler:

```
Begin
   Update bookkeeping
   If last RO copy
      Resume
End
```

Figure 2.4: Teapot pseudocode handlers

run. It sends invalidations to the holders of read-only copies and moves to the `RS_To_Excl` state to await acknowledgment of the invalidations. The second handler is run as each acknowledgement (`Put_RO`) is received. It notes the fact that a read-only copy has been returned and, if there are no more outstanding read-only copies, sends the exclusive copy to the original requestor.

Figure 2.4 shows the equivalent Teapot handlers. The handler for the `Get_Excl` request sends out invalidations, then calls `Suspend`. The `Suspend` call moves the protocol to the `RS_To_Excl` state and waits there until all invalidation acknowledgements have been received, at which point it sends off the exclusive copy and moves to the `Exclusive` state. The handler for the returned copies still performs bookkeeping duties but, as the acknowledgement for the last read-only copy arrives, calls `Resume` and returns control to the statement following the `Suspend`.

While an intermediate state is still required in which to wait for acknowledgements, the `Suspend` call has allowed all other action code for the transition from `RdShared` to `Exclusive` to be moved into a single handler. When implementing

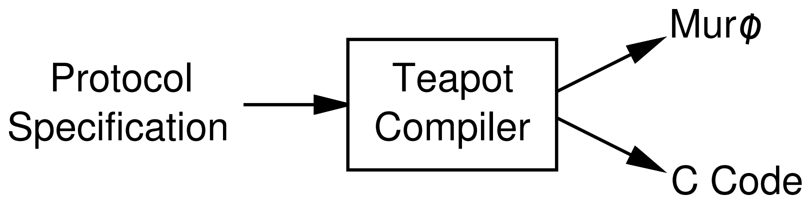


Figure 2.5: Teapot compilation paths

real, complex protocols this can significantly improve the readability and modifiability of the protocol code. It is not unusual to have handlers containing multiple `Suspend` calls in nested control structures. Without Teapot, the equivalent set of handlers is inscrutable.

Automatic Verification

Even with the improved programming environment provided by Teapot, protocols can be difficult to write correctly. Programmers must anticipate all possible messages that could arrive in a given state — including those due to reordering in the network — and handle them correctly. Finding protocol errors can be an extremely difficult and time-consuming task. Protocols contain complex timing-dependent paths, and bugs may not be repeatable. Even protocols that appear stable can harbor bugs waiting to be triggered by a particular sharing pattern or application. Teapot eases the debugging burden tremendously by exhaustively testing protocols for errors.

As shown in Figure 2.5, the Teapot compiler can turn protocol specifications into executable C code, or generate input to Murϕ [24], a formal verification tool. The verification process exhaustively explores a state space that is the cross product of all protocol, network, and data-structure states on each simulated processor. The size of the state space to be explored is therefore related to the complexity of the protocol and the size of the system configuration being verified. There are states in the verification space corresponding to all possible interleavings of access faults, message arrivals, and message reorderings in the network.

If verification is successful, it implies that all arriving messages have been anticipated and properly handled, and that the protocol will not deadlock on the system configuration tested. Practical considerations limit verification to configurations composed of two or three processors and one or two memory locations. The depth of the simulated network reordering can be controlled as well, and is typically tested for depths of up to two, which means that a given message can pass at most two other messages along the same network link. Verification does not currently test data values, though it would be possible to do so. While protocols can only be guaranteed to work on configurations as complex as those used during verification, in practice verifying on small configurations has been sufficient to eliminate (detectable) bugs from protocols running on real systems.

Chapter 3

Loosely Coherent Memory and C**

3.1 Introduction

This chapter describes Loosely Coherent Memory (*LCM*), a custom protocol that implements the semantics of a new parallel programming language C** [39] up to three times faster than other approaches.

Semantically, parallel tasks in C** execute *simultaneously* and *instantaneously*, so conflicting data accesses are impossible. Programmers need not consider potential interactions between tasks since no interaction is allowed. Implementing C** requires processors to keep local copies of modified data items to prevent changes from becoming globally visible until all parallel tasks have completed. Compilers can generate code to either explicitly copy modified items, or create renamed copies on-the-fly. But imprecise static analysis forces the explicit approach to copy a superset of the modified locations, and managing copies at runtime is complex and requires expensive runtime tests. LCM helps implement C** by allowing protocol-level copies of shared data to develop at runtime, and efficiently *reconciles* copies once all tasks have finished. Copying at the protocol-level is transparent (copies are at the same address as originals), and eliminates the need for runtime tests. Since it copies at runtime, LCM copies only modified locations.

LCM is an example of a larger class of Reconcilable Shared Memory (*RSM*) systems, which generalize the replication and merge policies of cache-coherent shared memory. RSM protocols differ in the action taken by a processor in response to a request for a locations and the way in which a processor reconciles multiple outstanding copies of a location.

The RSM model is described in Section 3.3 after a discussion of related work (Section 3.2). An overview of the C** language is given in Section 3.4. Several LCM implementations are described (Section 3.5), and all are formally verified (Section 3.6). Section 3.7 presents performance results and analysis.

3.2 Related Work

Relaxed consistency models [1, 26, 30] take advantage of the fact that global memory need not always appear consistent. Performance gains can be had by allowing incoherence to develop, but ensuring memory coherence at user-specified synchronization points. There is a similar lack of coherence in RSM between reconciliations, but the incoherence conforms to a semantics and can therefore be reasoned about. For example, LCM *requires* the predictable behavior of the incoherent memory for the correct implementation of C** semantics.

RSM shares with Munin [13] and TreadMarks [6, 35] the ability to adapt standard distributed shared memory policies to better suit an application. Both Munin and TreadMarks provide a set of fixed coherence mechanisms, each tailored for a specific sharing pattern. The user or compiler associates a coherence mechanism with each object. RSM allows dynamic schemes that can be applied at the cache block granularity as opposed to the language-object level.

In VDOM [29], memory objects are immutable, and an attempt to modify an object produces a new *version* of the object. It is related to RSM in that both systems allow multiple copies of memory items to develop. VDOM handles coherence at the language-object level, as opposed to RSM's finer-grained cache block level. It also uses a single, inflexible coherence mechanism based on object version numbers.

Like LCM, the Myrias machine [12] copied data on-the-fly to prevent interactions between parallel tasks. But the Myrias scheme was implemented in hardware, and copied data at the page granularity. Being hardware based, the copying and reconciliation policies were necessarily fixed.

The division of labor between the C** compiler and LCM is reminiscent of the techniques for stack and heap bounds checking [8] and concurrent garbage collection [7] in Lisp. With normal stop-and-copy garbage collectors, all pointers into the old heap can be replaced with pointers into the newly compacted heap. Concurrent collectors allow the computation to proceed, and attempt to copy objects from the old heap to the new without interruption. Virtual memory protections can be used to trap accesses into the old heap, at which point they can be transparently redirected to the appropriate object in the new heap.

3.3 Reconcilable Shared Memory

Reconcilable Shared Memory (RSM) is a family of memory systems that provides means by which a compiler can implement policies to control memory system behavior and performance. Both conventional cache-coherent shared memory and the LCM protocol fit within the RSM model. RSM assumes the same basic mechanisms as cache-coherent shared memory [55, 64] but generalizes the coherence policies. RSM systems differ in the action taken by a processor in response to a *request* for a location and the way in which a processor *reconciles* multiple outstanding copies of a location. Unlike most shared-memory systems, RSM places no restrictions on multiple outstanding writable copies of a block and permits non-sequentially consistent memory models.

Reconciliation of writable copies brings the copies' contents into agreement. It may also, depending on the reconciliation function, invalidate copies (remove them from processors' caches and memories). Reconciliation can return memory to a consistent state in which all copies of a location contain the same value. Reconciliation provides an opportunity to communicate values among processors and to perform computation on these values. An application program controls the request and reconciliation policies through memory system *directives*, which specify the policies for a region of memory.

Sequentially consistent, cache-coherent shared memory is a simple form of RSM. Since it fits within this model, it provides a natural default policy for a RSM system. Requests in these shared-memory systems return a copy of a block, subject to the guarantee that only one processor holds a writable copy at a time. In many systems [19, 42], a centralized directory controller records which processors hold copies of a location and invalidates outstanding copies upon request.

Reconciliation policies in these systems are also simple. Read-only copies are identical and so can be combined by a null reconciliation function. When a processor returns a writable copy of a block, its value is reconciled by making it the new value of the location. Update-based systems reconcile after modification to a shared location by assigning the new value to all copies.

3.4 C**

Much of the burden of programming in current parallel languages is due to programmers having to reason about interactions between concurrent processes. C**, a large-grained data-parallel programming language based on C++, eases this burden by providing a semantics in which parallel processes *cannot* interact. Processes can still collaborate to produce values via a rich set of reduction operations (including user-specified reductions), but the results of these reductions are not available until after all parallel tasks complete. During a parallel computation, no C** process can influence the state of another.

Parallelism in C** results from applying a *parallel function* across a collection of data called an *aggregate*. Aggregates look and behave like C++ arrays,

```
void stencil(parallel matrix &A) parallel
{
    // Pseudo variables #0, #1 specify position
    int x = #0, y = #1;

    // Average neighbors' values
    A[x][y] = (A[x-1][y] + A[x+1][y]
              + A[x][y-1] + A[x][y+1]) / 4.0;
}
```

Figure 3.1: C** parallel function

but form the basis for parallel functions. Applying a parallel function creates an asynchronously executed parallel function *invocation* for each element in an aggregate. These parallel function invocations appear to execute atomically and simultaneously, so there is no opportunity for them to interact. Modifications are private to an invocation, and cannot be seen by other, concurrently-executing invocations. After all invocations complete, the program's global state is updated by merging all private modifications. If two or more invocations modify the same location, C** specifies that exactly one modified value will be visible after the merge.

Figure 3.1 shows a C** parallel stencil function that averages its neighbors' values. When the parallel function is applied to an aggregate (the parallel argument `A`), a function invocation is called for each location in the matrix. Each invocation reads the values in neighboring matrix cells, then updates its own cell value. Without C** semantics, a programmer would not know whether neighboring values had been updated. Here, the reads are guaranteed to return unmod-

```
void new_stencil(parallel matrix &B, matrix &A) parallel
{
    // Pseudo variables #0, #1 specify position
    int x = #0, y = #1;

    // Read from A, write to B
    B[x][y] = (A[x-1][y] + A[x+1][y]
             + A[x][y-1] + A[x][y+1]) / 4.0;
}
```

Figure 3.2: Revised parallel function

ified values, since modifications made on neighboring cells are not visible until all invocations complete. Since `stencil` writes each data point exactly once, no modifications conflict and the merge phase collects the values assigned to each location.

3.4.1 Implementing C** Semantics

A compiler can always generate code that correctly implements the semantics of C**. Doing so requires identifying modifications to shared data and ensuring that the modifications are kept local until all invocations complete. For a program with regular structure like `stencil`, the analysis can be done statically. All locations in matrix `A` are updated during each call of the parallel function, so a complete copy of the matrix is required. Figure 3.2 shows a source-level rewriting of `stencil`, illustrating the introduction of a copy of matrix `A` to hide modifications. Each invocation reads from `A` as before, but writes new values to a matrix `B`. After the parallel function completes, `A` and `B` are swapped.

```
void dyn_stencil(parallel &A) parallel
{
    int x = #0, y = #1;

    if (foo) {
        // Must create copy of A[x][y]
        A[x][y] = A[x][y] + ((A[x-1][y] + A[x+1][y]) / 2.0);
    }
    if (bar) {
        // Have we already created copy?
        A[x][y] = A[x][y] + ((A[x][y-1] + A[x][y+1]) / 2.0);
    }
    // Have we modified A[x][y]?
}
```

Figure 3.3: Dynamic parallel function

For programs with dynamic access patterns, the compiler is forced to explicitly copy a conservative superset of the modified locations, since static analysis cannot precisely identify modifications. Alternatively, the compiler could generate code to perform the copying at runtime. By deferring copying decisions until execution, only as much data as necessary is copied.

Figure 3.3 shows a parallel function with dynamic behavior. Each invocation modifies a given location from zero to two times, depending on the values of a pair of conditional expressions. Prior to the first modification of `A[x][y]`, a local copy must be created. All subsequent references to `A[x][y]` — both writes *and* reads — must be preceded by tests checking whether a local copy exists. In the second assignment statement in Figure 3.3, the read of `A[x][y]` must be satisfied from the local, renamed copy if one exists. The write either modifies the previously-created copy or creates one and modifies it. Also, since each processor typically handles multiple function invocations, local copies must be managed such that modifications are not visible from one invocation to the next on the same processor. Finally, notice that the compiler cannot tell if `A[x][y]` has been modified by `dyn_stencil`. The compilation approach used for the earlier `stencil` function hid modifications by using a second array, but could only do so because analysis guaranteed every location would be modified. The array of modified values `B` will be incomplete unless unmodified values are explicitly copied from `A` to `B`.

3.5 LCM

Neither of the outlined approaches for implementing C** are particularly appealing. Generating code to perform explicit copying only works well for programs with static access patterns, and the dynamic copying method requires a complicated copy-management scheme and frequent runtime checks.

The coherence protocol underlying the C** code is *already* managing copies of shared data, and can be extended to incorporate the particular copying scheme required by C**. Protocol support for C** semantics has several advantages. Protocol-based schemes are dynamic, and only copy as much data as necessary to implement the semantics of C**. They also simplify the task of compilation, since the compiler can rely upon the protocol to implement the semantics. Copies maintained by the protocol can be referenced *transparently* — data resides at the same address whether it has been copied or not. Finally, runtime checks are

Directive	*Behavior*
`mark_modification(addr,size)`	Notifies LCM of pending modification
`reconcile_copies()`	Barrier; returns system to consistency
`flush_copies()`	Reconciles single invocation's writes

Table 3.1: LCM memory-system directives

eliminated, though the compiler must interact with the memory system prior to modifying shared data.

Loosely Coherent Memory (*LCM*) is an RSM memory system that implements the semantics of C**. The compiler uses RSM directives to identify memory accesses in a parallel function that possibly conflict. At these references, LCM copies the memory block containing the accessed location and makes it private to the invocation. If multiple invocations modify the same location, LCM creates local copies for each invocation. These multiple writable copies preserve the semantics of C**, even though shared memory as a whole is no longer consistent. When the parallel call terminates, LCM reconciles multiple versions of a block to a single consistent value.

LCM provides the C** compiler with the three directives shown in Table 3.1. The first, `mark_modification(addr,size)`, creates an inconsistent, writable copy of the memory block(s) containing locations `addr` through `addr+size`. The second, `reconcile_copies()` appears as a global barrier executed by every processor. When it finishes and releases the processors, the memory has been reconciled across all processors and is again in a consistent state. During the merge phase initiated by the directive, all modified blocks are flushed back to their home pro-

```
void stencil_wrapper(matrix &A)
{
  for(all invocations assigned to me)
  {
    set variables #0 and #1;

    // Function body:
    mark_modification(A[#0][#1],4); // LCM directive

    A[#0][#1] = (A[#0-1][#1] + A[#0+1][#1]
              + A[#0][#1-1] + A[#0][#1+1]) / 4.0;
    flush_copies();                 // LCM directive
  }
  reconcile_copies();               // LCM directive
}
```

Figure 3.4: Stencil code with LCM support

cessors to be reconciled. Outstanding read-only copies of these blocks are then invalidated throughout the system. The third directive, `flush_copies()`, performs a partial reconciliation by flushing a processor's modified cache blocks back to their home processors.

Figure 3.4 illustrates the directives' use. It shows a simplified version of the code generated by the compiler for the `stencil` function from Figure 3.1. Each invocation writes to `A[#0][#1]`, which is also read by its four neighboring invocations. Compiler analysis easily detects this potential conflict, which the C** compiler rectifies with `mark_modification` directive. The `flush_copies` directive removes modified copies from a processor's cache before another invocation starts. The `reconcile_copies` directive causes the memory system to reconcile modified locations and update global state to a consistent value.

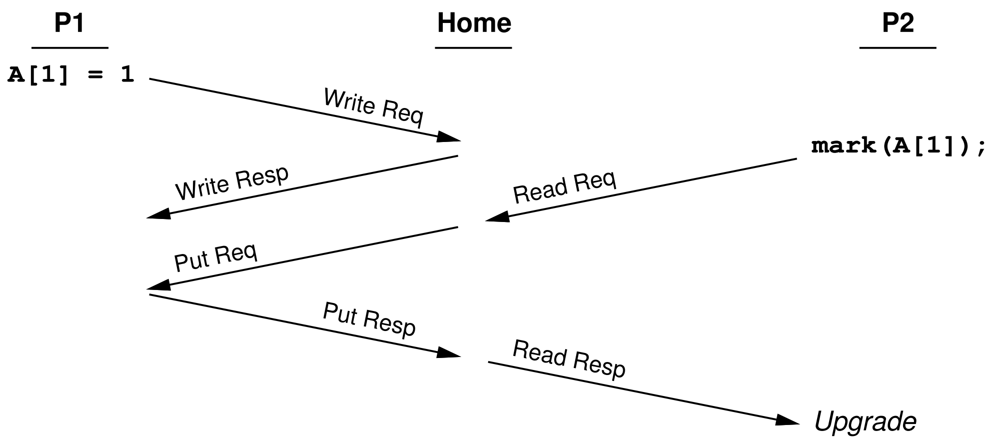


Figure 3.5: Requesting read-only block and upgrading

3.5.1 LCM Implementation

An LCM memory system was implemented using the Tempest interface [54]. It is based on an invalidation protocol similar to Stache [55], and provides cache-coherent shared memory as its default. Deviations from globally consistent memory come as a result of the LCM directive `mark_modification`, which creates local, writable copies of memory blocks. In general, these writable copies are produced by locally upgrading read-only blocks. If the processor invoking the directive has no copy of the block, it requests and upgrades a read-only copy. This has the advantage of causing the home to acquire and produce the most recent copy of the block, as shown in Figure 3.5.

Once a processor has called `mark_modification`, only read-only copies of the marked block exist as far as the home's directory is concerned. The home is guaranteed to have an up-to-date copy of the data, and can use this copy to satisfy read-requests from other processors. (The block was either in a read-shared state before a mark was performed, or the mark caused the home to acquire the most recent data.) The home must maintain an unmodified copy of the block for satisfying read requests, or the semantics of C** will be violated. Thus, if the home invokes `mark_modification`, a *clean copy* of the block is made in local memory before the block is modified and subsequent read requests are satisfied from the clean copy.

An invocation's modifications must remain accessible until it terminates, since processors have no way of retrieving or recreating modifications if data were prematurely flushed from the cache and returned home. The Tempest implementation on which LCM is built uses a processor's local memory as a large, fully associative cache, and prevents locally-modified blocks from being lost. When a modified cache block is selected for replacement (either because of a capacity or conflict miss), the block is moved to local memory. On a cache miss to the block, its value comes from memory, rather than its home processor.

It is equally important to ensure modifications are not visible to subsequent invocations on the same processor. In general, processors will handle many function invocations, and modified blocks must be removed and returned to the home after each invocation completes. The `flush_copies` directive (used in Figure 3.4) returns blocks marked during the current invocation to prevent them being read by later invocations. Modified blocks are merged with modifications from other invocations at the home, according to the combining operation specified by the *reconciliation function*. The merge results are written to an *accumulator copy* in

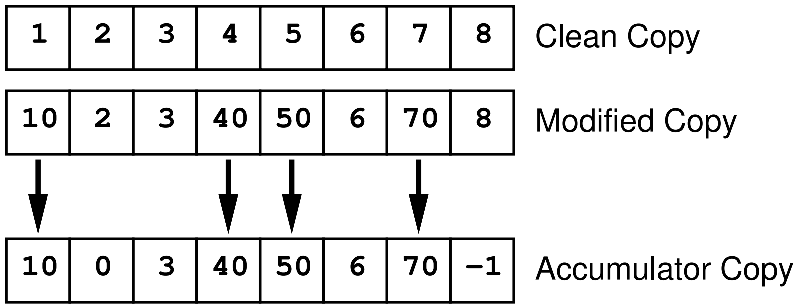


Figure 3.6: Reconciling a modified copy

local memory, where they stay until all invocations complete.

In Figure 3.6, a modified block is compared against the clean copy to find individual modifications. The four new values are then combined with those on the accumulator copy. The C** reconciliation function specifies that only one of a set of modifications to the same location is kept, so the the incoming modifications are simply written to the accumulator copy. Note that the entire modified copy could not be written to the accumulator without destroying previous modifications.

Processors call `reconcile_copies` to signal completion of their assigned invocations. Once all invocations complete and all modifications have been merged into their accumulator copies, the accumulators are written back to shared memory. They represent a block's final value after a parallel function call. Outstanding read-only copies of the updated blocks are then invalidated to ensure global memory is consistent. Once all invalidations are acknowledged, processors emerge from the call to `reconcile_copies`.

3.5.2 LCM-MCC

LCM correctly implements C** semantics, but can perform poorly for applications that share memory blocks across invocations. Consider the code in Figure 3.4. Compilers will typically allocate consecutive invocations to processors. Memory blocks would therefore be written by a series of consecutive invocations, since multiple values reside on each block. Enforcing C** semantics requires modifications be hidden from subsequent invocations, so the `flush_copies` directive removes modified blocks from the processor's memory after each invocation. But the next invocation modifies the same block, and it must be reacquired from the home.

Repeatedly flushing and retrieving blocks increases network traffic and causes delays while processors reacquire previously-flushed blocks. These effects can be reduced by keeping clean copies on both the home *and* remote processors. The first `mark_modification` call on a block creates a writable copy and a clean copy in local memory regardless of whether it is the home node for the block. Accesses by subsequent invocations can be serviced from the clean copy and do not traverse the network. Accumulator copies are kept on each processor as well, and the `flush_copies` directive locally reconciles modifications into the remote accumulator.

The LCM system maintaining remote clean and accumulator copies is called *LCM-MCC* (for Multiple Clean Copies) to distinguish it from the version that keeps a single clean copy at the home (*LCM-SCC*). Semantically, the two versions are equivalent, but LCM-MCC can be faster than LCM-SCC for applications that reuse memory blocks. But, the additional performance comes at the expense of

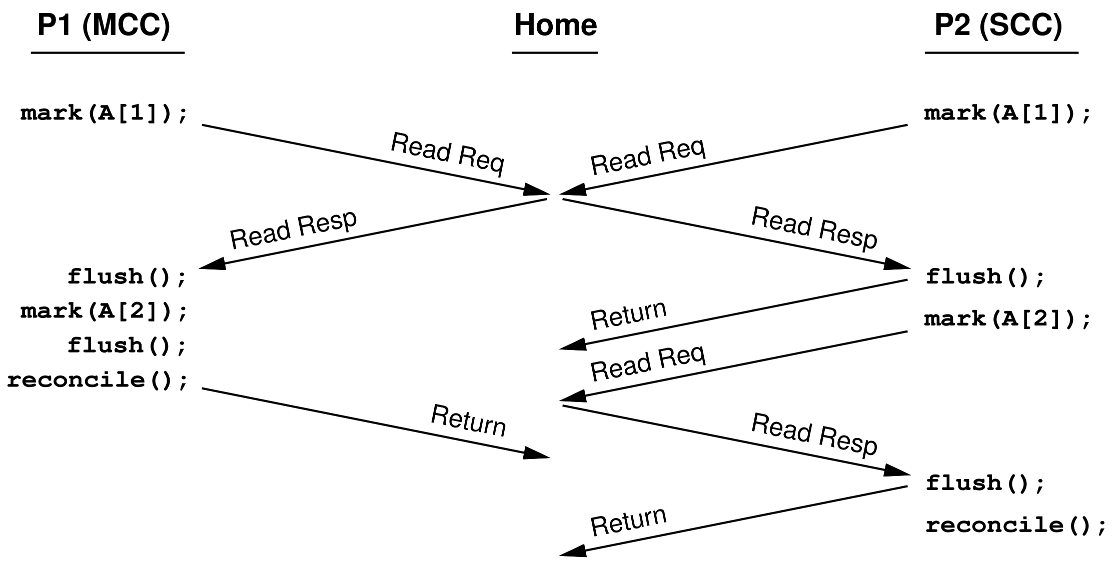


Figure 3.7: LCM-MCC and LCM-SCC

the memory required for the remote clean and accumulator copies. These memory overheads are detailed in Section 3.7.

Figure 3.7 illustrates the differences between the two LCM policies. Both processors modify a pair of consecutive memory locations (`A[1]` and `A[2]`). After modifying `A[1]`, processor `P2` flushes the modified block back to the home, forcing the subsequent reference to `A[2]` to retrieve the block. It is flushed again after writing `A[2]`. On processor `P1`, with the LCM-MCC policy, the flush is an entirely local process. When `A[1]` is flushed, a local accumulator copy is created and initialized with the block containing `A[1]`, and the local clean copy is used to refresh the cached data. The second flush, after modifying `A[2]`, is also a local process. Data is only transferred home at the start of the merge phase in LCM-MCC.

3.5.3 LCM-Update

Iterative applications like `stencil` often access the same locations across iterations. But, since processors flush all modified data home at the end of each parallel function call (under either LCM-SCC or LCM-MCC), blocks are not available for reuse and must be faulted back for the next iteration. During the merge phase, both LCM schemes record which processors held a given block, and can eagerly send updated data as it becomes available.

If an iterative application's access patterns change over time, update schemes run the risk of providing processors with updates for memory blocks no longer referenced. The update versions therefore implement a threshold scheme that determines whether to update or invalidate read-only copies based on the value of a counter. (Copies that have been written are clearly still in use, and are always updated.) After some number of updates, a read-only copy is instead invalidated. Only those processors still referencing the block will retrieve it on their next access. The threshold value is set via the `set_inval_thresh(int)` directive.

3.6 Verification

Teapot protocol specifications were written and verified for LCM-SCC, LCM-MCC, and the Update variations of each. (See Appendix A for state diagrams.) The verification process exhaustively explores a state space, the size of which is related to the complexity of the protocol and system configuration being verified. Systems with two or three processors, one or two memory locations, and varying

Version	*Proc.*	*Addrs.*	*Reord.*	*States*	*Rules*	*Seconds*
Stache	2	1	0	168	497	8
Stache	2	1	1	230	653	8
Stache	2	1	2	1,735	5,210	10
SCC	2	1	0	50,077	233,237	426
SCC	2	1	1	1,447,729	6,985,398	11,515
MCC	2	1	0	90,930	376,326	679
MCC	2	1	1	753,596	3,238,397	5,804

Table 3.2: Stache, SCC, and MCC verification results

amounts of network reordering were considered, though only configurations with two processors and one memory location could be exhaustively verified. While this is insufficient to guarantee that the protocol works correctly on larger configurations, it greatly increases confidence that it will do so. Verifying as much as possible of other configurations adds to this confidence, as it ensures that there are no errors in the explored space.

Data on completed verifications runs for LCM-SCC and LCM-MCC is given in Table 3.2. Verification data for the Stache protocol is included for comparison. The LCM protocols could only be exhaustively verified when network reordering was limited to zero or one.[1] For equivalent configurations, the Stache protocol requires far fewer global states be considered. The LCM protocol description uses a larger number of states, transitions, and message types than Stache, so more interactions must be explored. Also, in LCM, a larger variety of messages can coexist

[1]The network reordering number specifies the largest number of messages a given message can overtake in the network.

Version	*Proc.*	*Addrs.*	*Reord.*	*Thresh.*	*States*	*Rules*	*Seconds*
SCC	2	1	0	0	64,783	301,437	571
SCC	2	1	0	1	60,098	283,152	511
SCC	2	1	1	0	1,049,714	5,098,981	8,745
SCC	2	1	1	1	926,758	4,522,397	7,482
MCC	2	1	0	0	106,047	443,075	801
MCC	2	1	0	1	92,438	394,909	852
MCC	2	1	1	0	137,385	574,564	1,104
MCC	2	1	1	1	119,022	506,480	867

Table 3.3: Update verification results

in the network simultaneously, leading to a combinatorial explosion of verification states as the simulated reordering is increased. (Fewer messages can coexist under the LCM-MCC policy, and it is therefore affected less by increased reordering.) Finally, the LCM systems allocate and free block copies in local memory. No such mechanism exists in Stache, reducing the verification complexity.

Update versions of each LCM protocol were also verified, and the results are shown in Table 3.3. The threshold parameter in the table determines how many consecutive updates are sent to read-only holders of data. Values of zero (never send updates) and one (update every other iteration) were tested.

3.7 Performance

This section compares LCM to the compiler-only approach for implementing C** semantics, and demonstrates that programs with LCM support can run up to three times faster than those without. Guhan Viswanathan's C** compiler, as

Application	*Description*	*Problem Size*
Stencil	SOR over fixed grid	1024×1024 mesh, 15 iterations
Threshold	SOR with conditional writes	1024×1024 mesh, 15 iterations
Adaptive	SOR on adaptive mesh	64×64 mesh, 50 iterations
Unstructured	SOR on unstructured mesh	512 nodes, 2048 edges, 512 it.

Table 3.4: C** benchmark applications

the default, generates code to perform explicit copying based on (conservative) static detection of data conflicts. The compiler can also insert directives into the generated code to control an LCM memory system, and use LCM to ensure C** semantics at runtime. Identical source code was compiled and tested with both approaches.

The C** benchmarks and experimental setup are are described in Sections 3.7.1 and 3.7.2. Section 3.7.4 gives a detailed performance comparison between LCM and the compiler-only approach, after first describing LCM performance optimizations in Section 3.7.3.

3.7.1 Benchmarks

The four C** benchmarks listed in Table 3.4 were run with both compiler-only and LCM support. Each performs successive over-relaxation on a mesh, but has been tailored to explore a particular sharing pattern and behavior. Stencil operates over a fixed, 2D mesh of single-precision data. During each iteration, function invocations update mesh cells with the average of the four nearest neighbors' values. With eight mesh cells per 32-byte block, there is potential for reuse of cached data when processors handle consecutive function invocations. Threshold is similar to Stencil, but only updates cell values that *change*. The compiler cannot tell, statically, which values will be modified during a given iteration and must therefore copy the entire mesh to ensure C** semantics. The Adaptive benchmark operates on a 2D mesh of quad-trees that subdivide at runtime to capture regions of change in more detail. Since the compiler cannot predict node subdivision, it must conservatively copy the entire aggregate of trees. Also, since the structure of the trees change as nodes subdivide, the compiler-generated code must traverse both copies of the mesh between iterations and copy new subdivisions from the new copy to the old. Unstructured performs successive over-relaxation on an unstructured mesh. The connectivity of mesh nodes is determined by a data set read at runtime, and so cannot be known by the compiler. The lack of structure also gives little potential for reuse of cached data within an iteration.

3.7.2 Experimental Setup

All experiments were performed on a 32-processor CM-5 using the Blizzard-E [58] implementation of Tempest [55]. Five runs of each application were made on a given memory system, and the run exhibiting the smallest total execution time was selected. The LCM implementations tested were hand-coded versions written prior to the creation of the Teapot tool. These hand-coded protocols were the result of the lengthy and time-consuming performance tuning, and delivered better performance than the more recent Teapot-generated protocols. The compiler-only approach was supported by the hand-coded version of the Stache protocol.

Version	*Improvement*	*Tries/Send*	*Blocks/Msg*
LCM-MCC	1.00	25.9	1.0
+ bulk flush	1.30	9.7	63.9
+ bulk flush + asynch	1.44	6.1	63.9

Table 3.5: Bulk flush optimization (Stencil)

3.7.3 Performance Optimizations

The advantage of LCM-MCC over LCM-SCC is twofold: It allows reuse of memory blocks across invocations, and it lets processors locally combine modifications to the same block. But modifications are only transferred at the start of the merge phase, and the MCC policy can therefore cause increased network contention, since all processors return their modified data simultaneously. An optimization is to combine messages bound for the same destination into large bulk messages, reducing the total number of messages sent.

Table 3.5 shows the performance benefit of sending flushed data in bulk for Stencil. (Stencil benefits most from the LCM-MCC policy and bulk message sends.) Sending flushed data in bulk results in a speedup of 30%, and dramatically reduces network contention. A convenient way to assess contention is to measure the average number of attempts required to inject a message into the network. The data on the number of tries per send, shown in the third column, drops by 63%. On average, 63.9 blocks were transferred in each bulk message. Bulk messages can potentially contain up to 100 32-byte blocks in our system, but the averages are lower since not every bulk message can be filled.

Version	*Improvement*	*Tries/Send*	*Blocks/Msg*
LCM-MCC + bulk flush + asynch	1.00	6.1	—
+ update	1.08	12.5	1.0
+ bulk update	1.13	9.3	42.6
+ bulk update + asynch	1.15	8.9	42.6

Table 3.6: Update optimization (Stencil)

While sending flushed data in bulk reduces the number of messages sent, it also increases the time required for message handlers to process each incoming message. In general, handlers should run for as little time as possible, since the network is blocked during handler execution. A further optimization is to run the handler just long enough to copy bulk data into a buffer for later processing. The third line in Table 3.5 reports speedups for this asynchronous message reception/handling scheme. Overall execution time decreases by an additional 14%, and network contention drops to 6.1 tries per send.

These same optimizations apply to the sending of updates, since updates are all sent at the end of the merge phase. Performance data for LCM with updates is given in Table 3.6. The baseline in this comparison is the LCM-MCC system flushing modifications in bulk. Adding updates decreases the number of memory faults by a factor of 5.7 for Stencil, but increases network contention and only increases performance by an additional 8%. Sending the updates in bulk is an improvement, as is delayed processing of bulk updates. Taken together, these optimizations increase performance by an additional 15%.

In the following sections, the measured LCM-MCC implementation sends modified data in bulk with delayed processing. (This optimization does not apply to LCM-SCC, as it must flush data home immediately in response to the `flush_copies` directive.) Update versions of both LCM-SCC and LCM-MCC send updates in bulk and process them asynchronously. Blocks are invalidated after every 5 updates to ensure unnecessary updates are not sent.[2]

3.7.4 Performance Results

For each benchmark, C** semantics were implemented using either an LCM memory system or statically-generated copying code. The SCC, MCC, SCC-Update, and MCC-Update LCM policies were each tested, and data was taken for both static and dynamic task-scheduling schemes. In static scheduling, each processor is assigned the same set of parallel function invocations across iterations. The dynamic scheme attempts to improve load balancing by allocating tasks to processors on demand. Processors may therefore be assigned different invocations from iteration to iteration.

The performance results are summarized in Figure 3.8, with LCM results normalized to the compiler-only data. For all benchmarks except Stencil, and both scheduling schemes, programs with LCM support are faster than those using the statically-generated copying code. Improvements range from a few percent up to a factor of 1.5 for static scheduling, and from 1.5 to nearly a factor of 3 with dynamic scheduling.

[2] A wide range of thresholds were tried, and the default value of 5 was the best compromise between updating and invalidating.

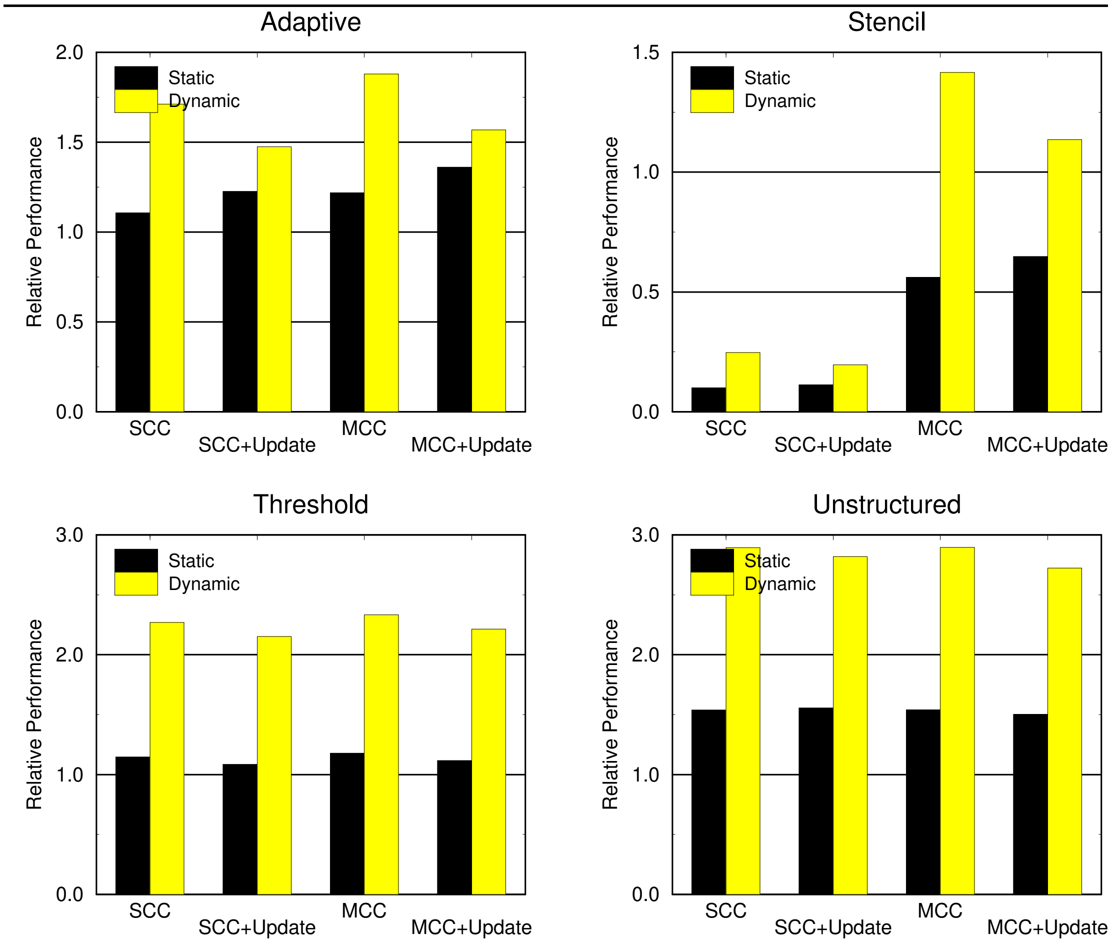


Figure 3.8: Improvements for C** benchmarks

Benchmark	*Faults (1000s), Static*			*Faults (1000s), Dynamic*		
	No LCM	*SCC*	*MCC*	*No LCM*	*SCC*	*MCC*
Adaptive	1,517	1,804	1,358	5,271	1,799	1,361
Stencil	675	17,330	2,154	4,182	17,455	2,282
Threshold	1,350	594	561	8,392	3,667	3,457
Unstructured	2,472	2,238	2,208	12,549	2,289	2,246

Table 3.7: Shared-memory access faults

LCM systems need only maintain copies of modified data, while the compiler-only approach copies all shared data in each of the four benchmarks. This gives LCM an advantage in the cases where benchmarks only modify a subset of shared memory during a given parallel function call. The compiler-generated copying code must update every aggregate location during each call, and can therefore generate a larger number of shared-memory access faults. But, once a block is faulted in, it can often be transparently reused by the copying code — both across invocations and across parallel function calls. LCM systems must remove modified blocks after each invocation and either flush them home or locally reconcile them. And, at the end of each parallel function call, LCM returns all modified data for reconciliation.

The balance between LCM systems and the compiler-only approach is influenced by the task-scheduling scheme, since static allocation lets the copying approach reuse more data across invocations and iterations than does dynamic scheduling. With dynamic scheduling, processors must acquire data for each new invocation handled. Both LCM-SCC and LCM-MCC are relatively indifferent to the scheduling scheme, since they must reacquire data across parallel function calls in either case.

Table 3.7 shows the total number of shared-memory access faults for each benchmark. With static scheduling, the compiler-only coping code can be seen to generate far fewer faults than with dynamic, while the LCM policies generate similar numbers of faults regardless of the scheduling scheme. LCM-MCC takes fewer faults in all cases than does LCM-SCC, but the difference is small

Benchmark	*Faults (1000s)*			*Tries/Send*		
	No Up.	*Update*	*Change*	*No Up.*	*Update*	*Change*
Adaptive	1,358	569	42%	6.5	9.2	142%
Stencil	2,154	376	17%	6.1	8.9	146%
Threshold	561	550	98%	1.4	1.5	107%
Unstructured	2,208	1,916	87%	2.3	3.1	135%

Table 3.8: Update statistics, static scheduling

for all benchmarks except Stencil, where blocks are modified up to eight times by consecutive invocations.

Update Effects

The data in Figure 3.8 helps explain the marginal improvement produced by LCM update schemes. The table shows updating's effect on network contention and access faults taken by each benchmark. Updates increase contention in all cases, and significantly reduce the total number of access faults for only two of the four benchmarks. Stencil sees the largest reduction of faults, as its repetitive access patterns are well served by the update mechanism. But, communication is hampered by the increased contention, and overall performance improvement is limited to 13%. Many of the faults in Adaptive cannot be eliminated by the update mechanism, since new memory is allocated by tree subdivisions during each iteration. Its performance increases by 10% with updates. In both Threshold and Unstructured, the number of faults during the computation phase is small compared to the overall total. In Threshold, for example, 509,000 of the 561,000 misses are due to initialization and a post-processing checksum phase. With

Benchmark	*Faults (1000s)*			*Tries/Send*		
	No Up.	*Update*	*Change*	*No Up.*	*Update*	*Change*
Adaptive	1,361	1,309	96%	7.0	16.6	237%
Stencil	2,282	2,155	94%	5.2	9.3	179%
Threshold	3,457	3,622	105%	1.6	1.6	100%
Unstructured	2,246	2,165	96%	2.4	3.0	125%

Table 3.9: Update statistics, dynamic scheduling

LCM support, mesh values are written only when their value *changes*. Relatively few locations change during a given iteration, so few faults result. The benefit of removing the remaining misses is outweighed by the overheads of the update mechanism and neither Threshold or Unstructured see performance improvements when updating.

Not surprisingly, sending updates when tasks are dynamically scheduled results in slowdowns for all benchmarks. Updated data is often not referenced since processors potentially handle different invocations across iterations. As shown in Table 3.9, the total number of faults is essentially unchanged by updates, but the network contention increases due to the update traffic.

Memory Overheads

Since LCM systems copy only modified locations, they can use significantly less memory than the compiler-only copying scheme. Figure 3.9 shows memory overheads for the LCM policies. LCM-SCC has overheads ranging from 20% to 102%, with the largest overhead on Stencil, where every shared location is modified. The compiler-only approach requires exactly 100% overhead for each benchmark, since

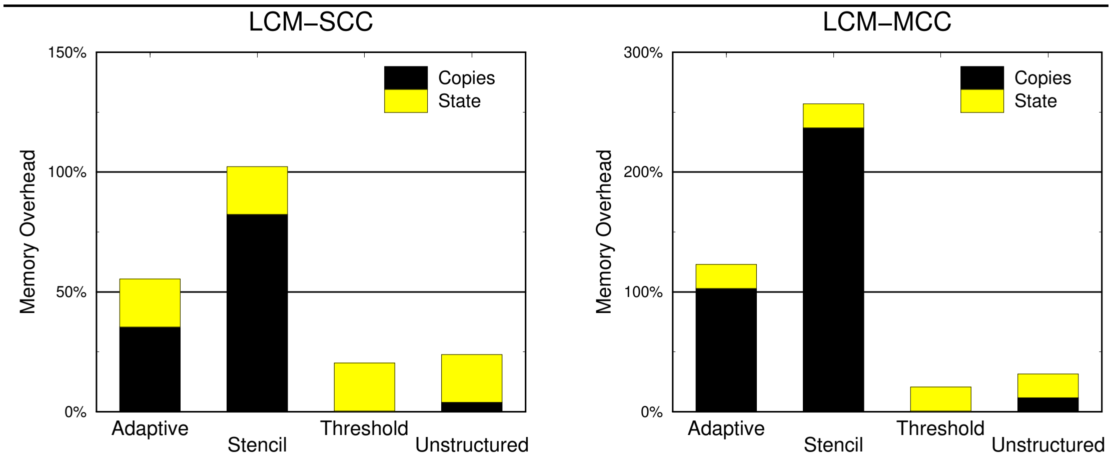


Figure 3.9: Memory overheads for LCM-SCC and LCM-MCC

it keeps two copies of all shared data. The overheads for LCM-MCC are roughly twice those for LCM-SCC, since it maintains multiple copies of each block.

LCM memory overheads come from two sources: Memory is consumed by copies of modified blocks, and is also required to record each block's state. In LCM, two additional pointers are kept per block, one each for the clean and accumulator copies. This extra state information is required for every block, whether copied or not. For benchmarks that modify relatively few locations (i.e. Threshold and Unstructured), the state overhead can be larger than the overhead due to block copies.

3.8 Conclusions

This chapter has described the design, verification, and implementation of a family of custom cache-coherence protocols that efficiently implement the semantics of a new parallel programming language C**. Semantically, parallel tasks in C** execute simultaneously and instantaneously, so data accesses cannot conflict. Implementing C** requires processors to keep local copies of modified data items to prevent changes from becoming globally visible until all parallel tasks have completed. These copies can be created by statically-generated code, or on-the-fly at runtime. Statically-generated code must conservatively copy a superset of the locations actually modified, causing an increase in the amount of required memory and a potential increase in the number of shared-memory access faults. Creating and managing copies at runtime is complex, and requires expensive runtime tests.

LCM helps implement C** by allowing protocol-level copies of shared data to develop at runtime, and efficiently reconciles copies once all tasks have finished. Copying at the protocol-level is transparent, and eliminates the need for runtime tests. On benchmarks without statically-analyzable access patterns, LCM gave better performance than the compiler-generated copying approach by up to a factor of three *and* used significantly less memory. The one benchmark for which precise static analysis could be performed, Stencil, is representative of a class of programs for which LCM support is unnecessary. Compilers can generate efficient code for these programs, and use LCM memory systems as a fallback position on code for which static analysis is imprecise.

Chapter 4

Other LCM Applications

4.1 Introduction

LCM was designed to implement C** semantics, but it can improve the performance of programs written in languages with more traditional semantics as well. LCM's update mechanism and its ability to efficiently reconcile multiple copies of data can significantly reduce overheads associated with fine-grained sharing of data. This chapter shows that LCM support improves overall performance of four C programs by up to a factor of 3, and can increase the performance of selected program phases by as much as a factor of 4.7.

In the previous chapter, the C** compiler inserted memory-system directives to control the behavior of LCM. Programmers did not participate in the collaboration between LCM and the compiler. This chapter shows that directives can be manually inserted into C programs to improve performance. While incorrect use of directives can lead to program errors, programmers with knowledge of a program's

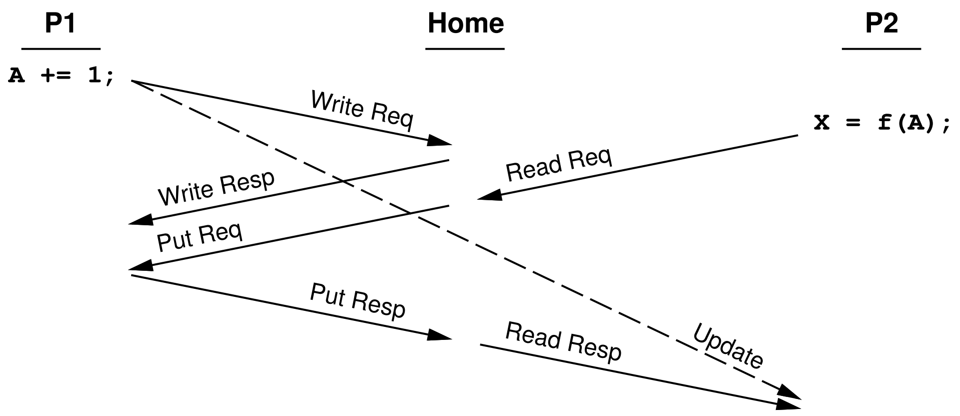


Figure 4.1: Producer-consumer sharing

communications and sharing patterns can use directives to tune memory-system behavior to fit the needs of an application.

4.2 LCM as an Update Protocol

It is well known that invalidation-based protocols perform poorly for programs with producer-consumer sharing. As shown in Figure 4.1, invalidation-based protocols require a series of messages to communicate a new value from a producer to a consumer. An update-based protocol saves time and reduces network traffic by directly communicating values from producers to consumers.

But, update-based protocols can be a poor match for programs without clearly defined producer-consumer sharing relationships, or with rapidly-changing relationships, as they can send updates to processors that no longer require them. The fixed protocol policy in hardware-implemented protocols results in an all-or-nothing decision between update and invalidation protocols, neither of which may be a perfect match for all program phases. Page-based DSM systems like Munin [13] and TreadMarks [6] are an improvement, as they allow protocol policies to be selected for individual program objects, but their large coherence granularity hinders performance for programs with fine-grained sharing. Tempest [54] allows programmers to use both update and invalidation protocols as required, and significant performance improvements have been obtained by tailoring update protocols to specific applications [28].

LCM provides both update and invalidation protocol policies at a fine coherence granularity, and does so without requiring new, application-specific update protocols for each program. By default, LCM uses an invalidation-based consistency scheme, but its bulk update mechanism can be used to efficiently apply an update-based policy on a block-by-block basis. Programmers or compilers can use LCM memory-system directives to mark modifications for which updates should be sent. Read-only copies of these locations throughout the system are updated when the `reconcile_copies` directive is executed. Modifications not preceded by marks will use the standard, invalidation-based consistency model. The next two sections demonstrate the effectiveness of the update scheme by using it to improve the performance of a pair of C-code benchmarks.

4.2.1 Chem

Chem is a computational chemistry application obtained from Iasonas Moustakis in the Chemical Engineering Department at the University of Wisconsin. As part

```
// Each processor modifies a portion of global vector A.
// New values are a function of entire vector.

FOR i in (my locations) {
    mark_modification(A[i]);  // prepare for an update
    A[i] = f(A);              // modify location
}
reconcile_copies();           // flush home and update
```

Figure 4.2: LCM-update support for Chem

of the computation carried out during each timestep, it solves a system of linear equations using the conjugate residual method. A global vector of solutions is shared by processors, each of which assumes responsibility for computing new values for a portion of the vector. Since computing these new values requires processors read the entire solution vector, processors are producers for values on their segment of the vector and consumers of the entire solution.

Figure 4.2 shows pseudocode for the equation-solving phase with LCM support. Before each processor creates new values for which updates are to be sent, the `mark_modification` directive obtains a local writable copy. This writable copy coexists with read-only copies in the system, as it did in C** applications, and does not cause their invalidation. After all locations have been written, `reconcile_copies` is called and modified memory blocks are flushed home for reconciliation. (The C** reconciliation function can be used, since each location is modified only once.) The LCM-update system then eagerly sends updated values to read-only holders of the modified blocks.

In C** applications, updated copies of a given block were typically sent to

Version	*Total Cycles*	*Improvement*	*Solve Phase*	*Improvement*	*Total Faults*
Chem	967M	1.000	558M	1.000	507K
Chem+LCM	665M	1.453	252M	2.219	344K
Chem+LCM+bcast	588M	1.644	197M	2.829	344K

Table 4.1: Summary of improvements for Chem

a handful of processors. In Chem, updates are sent to *all* processors. To increase performance, LCM's update mechanism was extended to more efficiently handle broadcasts of update data. Instead of sending updates to each processor in turn, a broadcast tree was formed, tailored to the timing constraints of the CM-5. Table 4.1 shows performance results for Chem with support from both the standard LCM update scheme, and the new broadcast mechanism. With broadcast updates, the overall computation speeds up by a factor of 1.6. The program phase solving the system of equations — the only portion with LCM support — improves by a factor of 2.8.

4.2.2 LCP

LCP, written by Satish Chandra and Steve Dirkse [21], solves the linear complementarity problem in parallel. Given a matrix M and a vector q, LCP finds a solution vector x such that $Mx + q \geq 0$. Like Chem, it divides the global solution vector among processors, and new values are a function of the entire vector. To amortize the cost of communicating new solution vector values throughout the system, processors in LCP refine a local copy of a segment of the vector before

Without LCM:

```
// Refine local solution

FOR l = 1 to 5 {
 FOR i in (my segment) {
  local_A[i] = f(local_A, A);
 }
}

// Write values to global copy
dt = 0.0;
FOR i in (my segment) {
 dt = max(A[i]-local_A[i], dt);
 A[i] = local_A[i];
}

// Find largest difference
LOCK();
  global = max(global, dt);
UNLOCK();
```

With LCM:

```
// Refine local solution
mark_modification(my segment);
FOR l = 1 to 5 {
 FOR i in (my segment) {
  A[i] = f(A);
 }
}

// Write values to global copy
dt = 0.0;
// Reconciliation finds dt and
// sends updates
reconcile_copies();

// Find largest difference
LOCK();
  global = max(global, dt);
UNLOCK();
```

Figure 4.3: Pseudocode for LCP, with and without LCM support

committing changes to the global copy. As local values are written back, processors monitor the difference between new and old solution vector values. The computation terminates when the largest difference is less than a given threshold.

Figure 4.3 gives pseudocode for LCP with and without LCM support. With LCM, processors need not maintain explicit local copies of solution vector segments. Copies are created at the protocol level when processors mark their portion of the vector. The reconciliation function for LCP is extended to find `dt`, by

Version	*Total Cycles*	*Improve-ment*	*Parallel Phase*	*Improve-ment*	*Total Faults*
LCP	2,024M	1.000	1,655M	1.000	1,419K
LCP+LCM	869M	2.328	475M	3.484	200K
LCP+LCM+bcast	686M	2.950	351M	4.715	200K

Table 4.2: Summary of improvements for LCP

observing the difference between new and old values as modifications are written to the accumulator copy. After all modifications have been combined, read-only copies in the system are updated.

Performance results for LCP are shown in Table 4.2. With LCM support and broadcast updates, LCP runs 3 times faster overall. But LCP initializes structures sequentially, and LCM only improves the performance of the parallel portion of the application, which runs 4.7 times faster with LCM support.

4.3 Efficient Reductions with LCM

LCM support can benefit a large class of applications by efficiently combining modifications both locally and across processors. Shared data items are typically modified by multiple processors during a parallel computation, causing write permission (and data) to transfer from processor to processor. Performance can often be improved by keeping processors' modifications local, and combining them at the end of the parallel computation.

Figure 4.4 illustrates two processors modifying the same location `A`. Processor

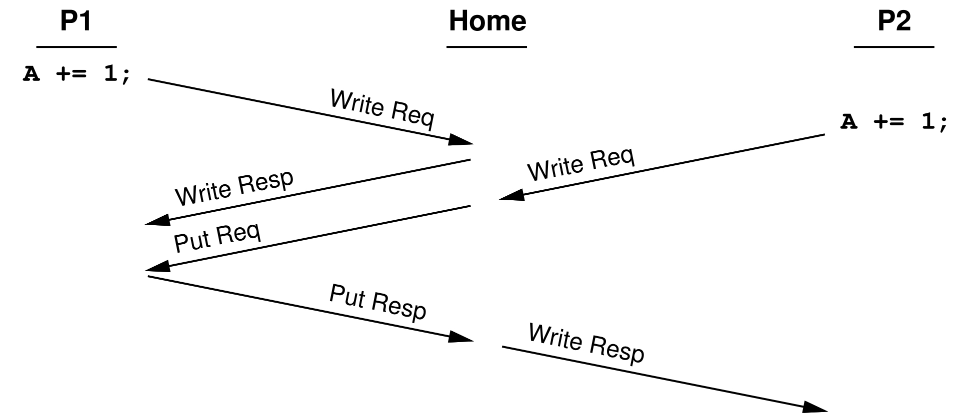


Figure 4.4: Processors competing to modify location

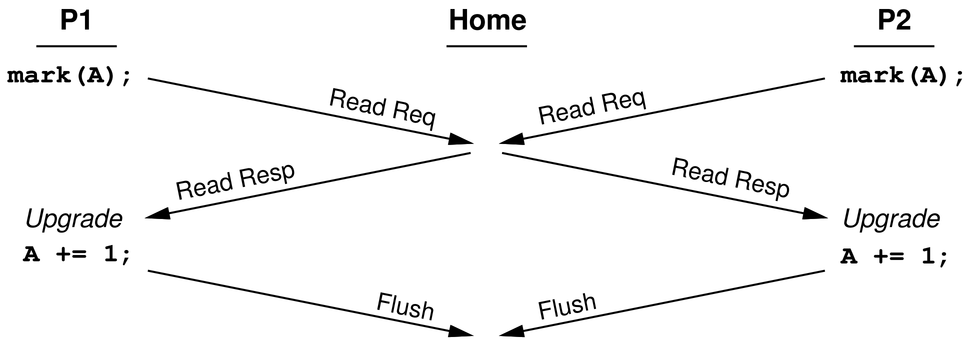


Figure 4.5: Multiple modifications with LCM

P1 first increments its value. Before P2 can contribute to A's value, the home must remove the block containing A from P1. The writes are serialized since only one processor can modify A at a time. In Figure 4.5, LCM support allows each processor to obtain a local, writable copy. Writes to these copies can be performed *simultaneously*, and without write permission moving between processors. During the reconciliation phase, local copies are combined and the contributions from each processor summed.

This scheme is similar to that used by the multiple-writer protocols in Munin and TreadMarks. While they allow simultaneous writes to the same coherence unit (page), they currently have no support for merging modifications to the same *location*, and therefore cannot perform reductions in the same manner as LCM. As will be seen in the following sections, these reductions are a powerful technique for improving the performance of applications with fine-grained sharing.

4.3.1 Water

Water is one of the Splash [60] benchmarks, and simulates interactions between molecules in a body of water. Processors are assigned a fraction of the simulated molecules, and are responsible for computing interactions between these and all others in the system. A pseudocode outline of the program is shown in Figure 4.6. Without LCM support, processors lock molecules involved in an interaction, acquire write permissions, and modify their global states. The LCM code uses `mark_modification` to obtain a local copy of each molecule before changing its value. Processors can therefore modify molecules simultaneously, and need

```
Without LCM:

FOR m in (my molecules) {
  FOR n in (m ... m + N/2) {
    compute interaction
    LOCK m;
      m += ...;
    UNLOCK m;
    LOCK n;
      n += ...;
    UNLOCK n;
  }
}
```

```
With LCM:

FOR m in (my molecules) {
  FOR n in (m ... m + N/2) {
    compute interaction
    mark_modification(m);
    m += ...;

    mark_modification(n);
    n += ...;
  }
}
reconcile_copies();
```

Figure 4.6: Pseudocode for Water, with and without LCM support

not compete for write permission. Once all interactions have been computed, modifications are flushed and summed at the home node.

With or without LCM support, processors modify their molecules repeatedly, as each is involved in interactions with every other molecule in the system. The relative benefits of LCM increase with the number of modifications to each molecule, as local writable copies can be reused at no expense while the base application must potentially reacquire write permission for each modification. This weakness was recognized by the authors of Water, and a revised version (Water2) was released that locally accumulates modifications and combines them after performing all interactions.

Table 4.3 compares the performance of Water with LCM support to both the original Water code and the improved Water2. Overall, LCM support improves

Version	*Total Cycles*	*Improve-ment*	*Parallel Phase*	*Improve-ment*	*Total Faults*
Water	4,278M	1.000	2,919M	1.000	5,138K
Water2	1,396M	3.064	747M	3.908	1,092K
Water+LCM	1,579M	2.709	697M	4.188	574K

Table 4.3: Summary of improvements for Water

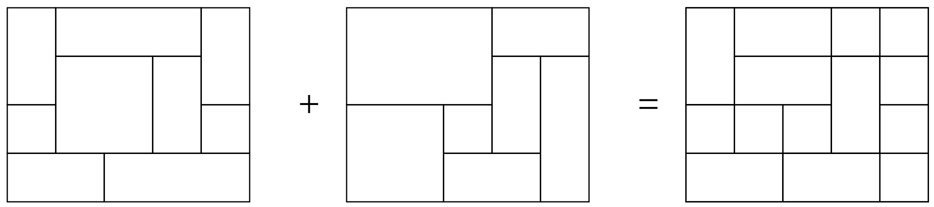


Figure 4.7: Intersection of two polygon maps

performance by a factor of 2.7, with an improvement of 4.2 during the parallel phase. Water does extensive sequential initialization of its data structures (for which LCM support offers no performance improvement), so improvements for the parallel phase are a better measure of LCM's effectiveness. The original Water code with LCM support is slightly faster than even the improved Water2 code, since it need not keep explicit local copies of molecule data. The resulting decrease in memory access faults gives Water+LCM a slight performance advantage.

```
Without LCM:

FOR p in (my polygons) {
  find overlapping cells
  FOR c in (cells) {
    LOCK c;
      insert(p, c);
    UNLOCK m;
  }
}
```

```
With LCM:

FOR p in (my polygons) {
  find overlapping cells
  FOR c in (cells) {
    mark_modification(c);
    insert(p, c);
  }
}
reconcile_copies();
```

Figure 4.8: Pseudocode for Overlay, with and without LCM support

4.3.2 Overlay

Overlay, written as a programming exercise for a book on parallel programming languages [41], computes the geometric intersection of a pair of rectangular polygon "maps". Each map covers the same geographical area, and is composed of a set of non-overlapping polygons. Figure 4.7 shows two such maps, and their intersection. Map intersections are computed in two phases: First, each input map's area is partitioned into cells, and lists containing all incident polygons are constructed for each. These cells help reduce the number of comparisons performed in the second phase, and form the basis of parallelism in Overlay. Next, for each cell, polygons from the first input map are tested against those from the corresponding cell in the second. As overlaps are found, polygons are created and added to the solution.

The partitioning phase has properties similar to the interaction of molecules in Water: Each polygon from an input map "interacts" with one or more partition cells, and causes a change in their state. Pseudocode for Overlay is shown in Figure 4.8. Without LCM support, each cell overlapped by a polygon `p` must be locked as `p` is inserted into the cell's polygon list. LCM allows processors to obtain local writable copies of each cell list, and perform the insertion simultaneously with insertions from other processors.

Previous applications of LCM have combined simple, scalar data types. Here, the reconciliation function must combine linked lists during the merge phase. Lists in Overlay are represented by a structure containing pointers to the head and tail elements. A pair of lists can therefore be concatenated without having to traverse either. The reconciliation function for Overlay (Figure 4.9) is passed pointers to clean, modified, and accumulator copies of a memory block. It treats data on these blocks as list structures, and concatenates lists from the modified copy to those on the accumulator copy. Figure 4.10 illustrates the process graphically.

LCM allows processors to transparently construct local lists of polygons for each cell, and efficiently combine these lists during the merge phase. The base Overlay code was improved to explicitly construct local polygon lists as well, but the simpler code with LCM support outperforms the new code as it manipulates less data and therefore takes fewer memory access faults. Table 4.4 gives performance improvement results for the overall execution times, and the partitioning phase for which LCM support has been added. LCM improves the partitioning by a factor of 4.6 over the code explicitly constructing local lists of polygons. (The improvement over the base code that locks each cell and directly inserts polygons is a factor of 12.6!)

```
void rec_fn(polyList_p clean, polyList_p dirty, polyList_p accum)
{
  int i;
    // for each cell list on the memory block
    for (i=0; i<(STACHE_BLK_SIZE/sizeof(polyList_s)); i++) {
      // if new list different, concat dirty+clean
      if (clean->head!=dirty->head) {
        dirty->tail->next = accum->head;
        accum->head = dirty->head;
        if (accum->tail == NULL)
          accum->tail = dirty->tail;
      }
      clean++;  dirty++;  accum++;
    }
}
```

Figure 4.9: LCM reconciliation function for Overlay

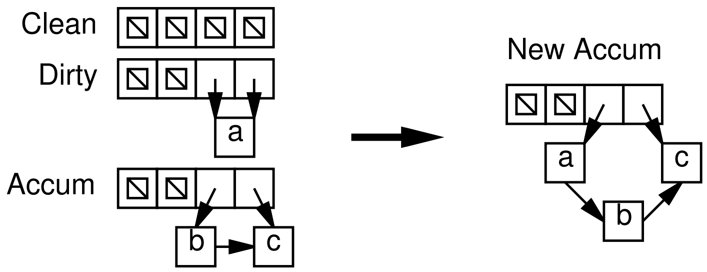


Figure 4.10: Merging polygon lists

Version	*Total Cycles*	*Improvement*	*Partition Phase*	*Improvement*	*Total Faults*
Overlay	341M	1.000	287M	1.000	61K
Overlay+LCM	123M	2.779	62M	4.628	39K

Table 4.4: Summary of improvements for Overlay

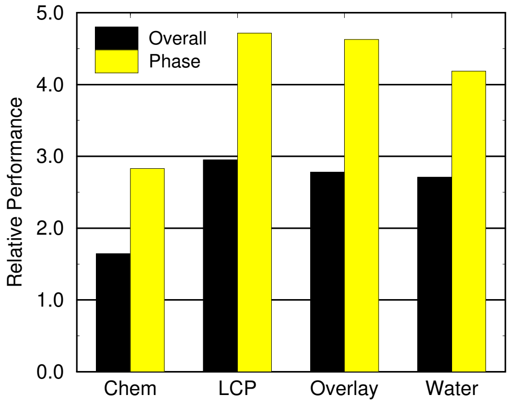


Figure 4.11: Summary of application improvements

4.4 Conclusions

This chapter shows LCM support can improve the overall performance of C-code applications by up to a factor of 3, with performance improvements as large as a factor of 4.7 in the program phases aided by LCM. (Results are summarized in Figure 4.11.) For two of the four benchmarks examined, these improvements are a result of LCM's ability to selectively apply an update-based coherence policy instead of the default invalidation-based scheme. The remaining two applications benefit from LCM's transparent copying properties and efficient reconciliation.

In each of the benchmarks, LCM memory-system directives were added manually to correctly-functioning programs. This incremental approach to performance improvement is attractive, but requires that programmers understand the communications and sharing patterns in their programs well enough to correctly insert directives, as their misapplication can introduce errors. Potentially, compilers or other automated tools could be extended to insert directives, eliminating the dangers of their misuse.

Chapter 5

Protocols for Detecting Data Races

5.1 Introduction

This chapter describes the design and implementation of a family of custom cache-coherence protocols that perform on-the-fly detection of apparent data races for programs with barrier synchronization. Overhead in execution time for these protocols are shown to range from zero to less than a factor of three over a set of benchmarks — a significant improvement over slow-downs of three to six for Dinning and Schonberg [25], and five to 30 for Perković and Keleher [53]. Hood, Kennedy, and Mellor-Crummey [33] have lower overhead than the protocol-based techniques, at approximately 40%, but require compiler involvement.

Efficient detection of data races is possible on DSM systems because a mechanism is already in place to invoke the coherence protocol in response to shared-memory accesses. The protocol can be extended to maintain access histories, detect concurrency, and watch for data races. The key strength of protocol-based scemes is that they are completely independent of program source code. Race detection can be performed on programs written in any language, and on library routines for which the source may not be available.

Sections 5.2 and 5.3 provide context for this work by introducing previous work on the formalization and detection of race conditions. The custom protocols are described in Section 5.4, and Section 5.5 details the efforts to formally verify them. Section 5.6 gives performance results and analysis.

5.2 Background

A race condition arises in a shared-memory parallel program when accesses to shared memory are not properly synchronized. Since this lack of synchronization can lead to programs that behave unpredictably, it is important to be able to detect and report these conditions. The race detection literature has used a variety of terms to describe race conditions, but this thesis follows Netzer and Miller [48, 51].

5.2.1 Types of Race Conditions

Netzer and Miller [50, 51] recognized two fundamentally different types of races. *Data races* are failures in nondeterministic programs, and occur when critical

```
Source Code:                    Process 1:        Process 2:

FOR i = 2,3          ==>        lock(L)           lock(L)
  A(i) = A(i-1)                   A(2) = A(1)       A(3) = A(2)
END                             unlock(L)         unlock(L)
```

Figure 5.1: Example of a general race

sections are not executed atomically. *General races* are failures in programs that are intended to be deterministic, and occur when the execution order of certain accesses is not guaranteed. Figure 5.1 shows an example of a general race. A loop with two iterations is being (incorrectly) parallelized by distributing the iterations across processors. The `lock` and `unlock` calls surrounding the loop bodies guarantee they execute atomically, but the synchronization does not *order* the execution of the iterations. Here, a loop-carried dependence requires that the first process execute before the second. Thus the program exhibits a general race even though there are no data races. Note the example would contain a data race as well if one or both of the calls to `lock` were missing, since the read and write of `A(2)` could execute concurrently.

Two variations of each type of race exist: *feasible* and *apparent*. A feasible race is one that occurs in some realizable program execution. Unfortunately, as Netzer proves [48], finding feasible races of either type is NP-hard for all types of synchronization. Apparent races occur when a race is detected in an execution permitted by the synchronization of the program (ignoring constraints caused by data dependences). The set of apparent races is an approximation to that of feasible races, since some of the executions exhibiting races may not be realizable. Finding apparent races is still NP-hard for programs using synchronization strong enough to implement two-process mutual exclusion, but is tractable for weaker synchronization.

For data races, a third type exists. *Actual* data races occur when atomicity of a critical section is violated. The set of actual races is a subset of the apparent races since it captures races that actually occurred in an execution, and not those that had the potential of occurring. Actual general races do not exist. Intuitively, this is because one cannot detect a general race without comparing the exhibited program ordering against all possible executions.

The protocols developed in this chapter find and report apparent data races on-the-fly. The details are discussed in Section 5.4.4. Apparent races are more useful from a diagnostic standpoint, because, while they can report infeasible races, they report accesses that *could* have executed concurrently and not just those that did. A single execution without apparent races therefore guarantees that all executions with the same data dependences will be race free.

5.2.2 Detecting Races

Accesses to a common memory location by a pair of processes are said to *conflict* if at least one of them is a write. Race conditions are the result of conflicting accesses made by blocks of code that could potentially be unordered. Detecting data races therefore requires determining whether or not program synchronization orders pairs of code blocks. In general, statically determining which blocks of code could

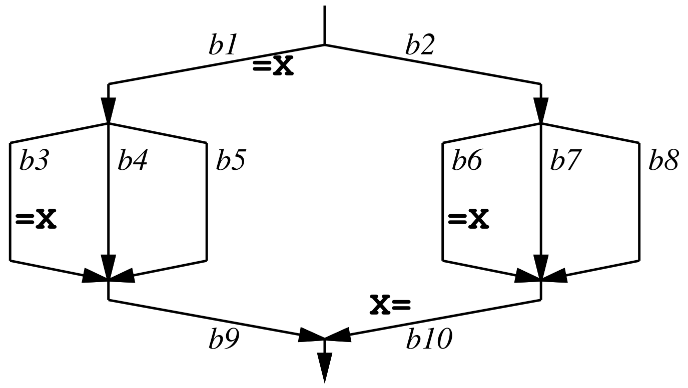


Figure 5.2: Partial order execution graph

execute concurrently is undecidable, since it requires precise information about control flow and dependences, but the ordering information can be approximated if one assumes all paths through a program are possible. This potentially lists blocks as being unordered that could never execute concurrently. (A pair of blocks found to be concurrent could be preceded by conditionals that ensure only one of them ever runs, for example.) It is exactly this inaccuracy that leads to the distinction between feasible and apparent races: Races declared on the basis of approximated concurrency data (apparent races) might not be realizable.

The code ordering constraints can be represented by a partial order execution graph (*POEG*), where edges represent blocks of code and vertices represent coordination events. Figure 5.2 shows a POEG corresponding to a program with two levels of fork/join parallelism. The write of `X` in code block *b10* conflicts with the reads of `X` in *b1* and *b3*. Races exist between these pairs of blocks since the program imposes no orderings between them. There is no race between *b6* and *b10*, since they are ordered by synchronization in the program (the join at the end of $b6$).

Three basic approaches have been used to detect races. Static techniques [4, 9, 16, 27, 62] examine the text of a program and use static analysis to approximate the shared-memory locations accessed by each code block. This information, combined with the concurrency information in the POEG, detects apparent races by finding pairs of unordered blocks making conflicting accesses to common locations. Static methods are necessarily conservative, since information on accessed memory locations is not precise, and the resulting spurious race reports can overwhelm users [33, 46].

Post-mortem [5, 23, 49] and on-the-fly methods [25, 33, 46, 52, 59, 61] improve race-detection accuracy by instrumenting programs and collecting information from actual executions. This information is either analyzed off-line, in the case of post-mortem techniques, or during execution, in on-the-fly methods. The monitored programs generate complete information about memory locations accessed, and can detect accesses by unordered blocks of code. Both approaches find apparent, and not necessarily feasible, races.[1] The primary drawback of post-mortem schemes is the large amount of storage required for complete traces of long-running programs, leading many to prefer the on-the-fly approach.

---

[1] Netzer and Miller, however, describe a post-mortem analysis technique that can improve the accuracy of the detected races by ruling out some apparent races that are artifacts of earlier races or prohibited by program dependences. [50].

5.3 Related Work

Traditional on-the-fly approaches maintain a history of the blocks that have accessed each shared variable. Code is added to the application program so that at each reference to shared memory a block compares its label against all labels in the access history. A race has occurred if the current access conflicts with a previous access by a concurrent block. For example, in Figure 5.2, assume the read of `X` in *b3* occurs before the write in *b10*. The read leaves its label in the access history for `X`. When the write is performed, block *b10* examines the access history and discovers that *b3* has accessed `X`. A race is declared once it is determined that *b3* and *b10* are unordered in the POEG.

On-the-fly methods must encode concurrency information such that it can be quickly consulted at each application read and write. This is usually achieved through block labeling schemes that reflect the position of each block in the POEG. A label comparison can then determine the ordering of a pair of blocks.

As described, the on-the-fly approach cannot handle programs with pairwise synchronization, since the orderings it introduces are not encoded in the block labels. (The orderings imposed by pairwise synchronizations cannot be known at compile time, and so cannot be included.) Dinning and Schonberg [25] associate a *coordination list* with each block that records information about immediate ancestors introduced by pairwise synchronization. A block *b1* is concurrent with *b2* if their labels reveal them to be unordered *and* none of the labels in *b1*'s coordination list are ordered with *b2*.

All on-the-fly race-detection schemes monitor memory accesses at runtime.

They differ in the synchronization primitives they allow, and the way they detect concurrency. This thesis introduces the term *system-level* to describe techniques, including the race-detection protocols, that discover concurrency from system-level information. In these systems, accesses to the same location are concurrent if, during execution, they are not separated by a synchronization event.

System-level schemes can detect actual or apparent data races, depending upon how they determine concurrency. Apparent races are found if the system declares unordered accesses to be potentially concurrent. If only the accesses that actually overlap are detected, the system finds actual races. Apparent races are more useful from a diagnostic standpoint, because, while they can report infeasible races, they report accesses that *could* have executed concurrently and not just those that did. Thus, a single execution without apparent races guarantees that all executions with the same data dependences will be race free, since no accesses between barriers conflict.

Since system-level approaches detect concurrency directly, there is no dependence on the POEG or program source code. Race detection can be performed on programs written in any language, and on library routines for which the source may not be available. However, the lack of source-level information also means that system-level schemes cannot take advantage of optimizations requiring knowledge of the source, such as removing accesses that are statically known to be ordered from race-detection consideration.

5.3.1 Traditional Approaches

Dinning and Schonberg [25] have implemented a general scheme for detecting apparent data races. They support both fork/join and pairwise synchronization, and obtain concurrency information from the POEG. The race-detection protocols only support barrier synchronization. Over a set of four benchmarks, Dinning and Schonberg report program slow-downs of from three to six using access histories limited to only one or two entries — roughly twice as slow as the protocol-based approach.

Mellor-Crummey [46] describes a method for encoding the POEG, *offset-span labeling*, that has improved space and time bounds for programs that do not use pairwise synchronization, but gives no performance results. Hood, Kennedy, and Mellor-Crummey [33], present a technique for detecting apparent data races in Fortran programs that use barriers and structured synchronization based on ordered sequences. They keep only one entry in the access histories, and use static analysis to reduce the number of monitored shared variables. Their slow-downs are roughly 40%, but the technique requires compiler support.

5.3.2 System-Level Approaches

Perković and Keleher [53] have implemented system-level race detection in CVM, a page-based release-consistent DSM. Systems that implement release consistency must maintain ordering information that enables them to make a constant-time determination of whether two accesses are concurrent. Perković and Keleher extended the DSM to collect information about referenced locations and check at barriers for concurrent accesses to common locations. Pairwise synchronization is handled as well as fork/join and barrier, since the DSM system must already be aware of all forms of program synchronization. On a set of four benchmarks, slow-downs ranged from a factor of more than five to almost 30.[2] The overheads for the custom protocols in this chapter are less than a factor of three.

Being a page-based approach, Perković and Keleher's system works well on a smaller set of applications than the race-detection protocols, since page-based DSM systems do not efficiently support applications with fine-grained sharing. Also, their implementation can miss some data races since they detect writes by discovering modified *values*. When a page of data is returned home, the system compares it to an unmodified copy. Differences are flagged as writes, so writes that do not change the contents of a location are missed. The custom protocols in this chapter are more precise, as they detect all writes.

Perković and Keleher check for data races only at barriers. There can be a potentially large lag between the detection of a race and the point at which it actually occurred, limiting the amount of information available for describing the race. My protocols detect races as they occur, and have perfect knowledge of at least one of the conflicting references.

The work in this thesis is most closely related to a hardware-based cache coherence protocol for CCNUMA machines that Min and Choi [47] designed but never implemented. Like the race-detection protocols, they limit access histories

[2] They apparently have improved numbers in a version of the paper accepted to OSDI, but not yet available.

to a single entry and do not support pairwise synchronization. Their scheme can miss shared-memory accesses unless the compiler organizes shared data such that at most one word per cache block is used. This restriction would cause an unacceptable increase in the memory requirements of a program, and waste precious bandwidth as cache blocks containing a single word of useful data are communicated between processors. The protocols in this chapter allow arbitrary data placement by keeping blocks invalid even after fetching data in response to a fault. Every reference to an invalid block invokes the protocol, so the protocol sees all references.

5.4 Design

Data-race detection requires two components: knowledge of accessed locations, and a means of determining concurrency. Approaches for monitoring accessed memory locations are discussed in Section 5.4.1, and the method used to determine whether accesses are concurrent is described in Section 5.4.2. Section 5.4.3 shows how the two are combined to detect races.

5.4.1 Monitoring Accesses

A coherence protocol can be extended to update an access history as part of the actions it performs in response to an access fault. This provides a method for monitoring shared-memory accesses. Unfortunately, in existing systems, a protocol is not aware of *every* reference. Once a faulting access is handled, subsequent

Source Line	*Cache Behavior*	*Protocol Behavior*
`A(1) = ...`	Write fault	Detects write
`A(2) = ...`	Cache hit	Does not see
`A(3) = ...`	Cache hit	Does not see
`A(4) = ...`	Cache hit	Does not see

Table 5.1: Example of missed accesses

accesses to the block can proceed without protocol involvement. This is by design, since unnecessary invocation of the protocol decreases performance. But race-detection protocols potentially need to monitor every access or they can miss data races. Table 5.1 illustrates missed accesses by showing how four consecutive lines of source code affect the cache and protocol. The write to `A(1)` causes a fault and invokes the protocol, which obtains a writable copy of the block. Array locations `A(1)`–`A(4)` are located on the same (now writable) cache block, so subsequent writes hit in the cache and go unnoticed by the protocol. Any data races involving the final three writes would therefore be missed.

My solution is to keep blocks invalid even after fetching data in response to a fault. Every reference to an invalid block invokes the protocol, so the protocol sees all references. As is shown in Section 5.6, the resulting program slow-downs are quite reasonable. *Note that* cache *and* protocol *access permissions for a given block are now independent.* A processor can hold (protocol) permission to write a block even though the block is cached in an invalid state.

Table 5.2 shows the same source lines, now with the cache access permissions and protocol states encountered during writes to `A` under the new scheme.

Source Line	*Cache Acc. Perm.*	*Protocol Behavior*
`A(1) = ...`	Invalid	Block is invalid
`A(2) = ...`	Invalid	Have exclusive copy
`A(3) = ...`	Invalid	Have exclusive copy
`A(4) = ...`	Invalid	Have exclusive copy

Table 5.2: Cache access permissions and protocol states

The protocol requests exclusive access to the block when the write to `A(1)` is attempted. Previously, the protocol upgraded the cache access permission to writable once it obtained the exclusive copy, and allowed the write to proceed. This caused later writes to be missed. The new scheme leaves the block containing `A(1)`–`A(4)` in an invalid state throughout the example. Thus, each write invokes the coherence protocol and no accesses are missed. Note that writes to `A(2)`–`A(4)` can be handled locally since the processor still holds the exclusive copy of the block. The protocol simply records the access, then allows it to proceed.

Implementing this scheme requires that blocks tagged invalid by the protocol hold data and be accessible by the processor. Currently, not all Tempest implementations guarantee that data values are maintained on invalid blocks.[3] One could still implement the race-detection protocols on non-data maintaining Tempest systems by keeping *copies* of invalid cache blocks in local memory and using them to satisfy references. Also, there is discussion of extending the Tempest interface to include access permissions that cause the protocol to be invoked re-

[3]The race-detection protocols represent a class of protocols that are difficult to implement without such guarantees, and make a strong argument for all Tempest systems being data-maintaining.

Source Line	*Cache Acc. Perm.*	*Protocol Tag*	*Protocol Behavior*
`A(1) = ...`	Invalid	Invalid	Block is invalid
`A(2) = ...`	Writable	Invalid	Have exclusive copy
`A(3) = ...`	Writable	Invalid	Have exclusive copy
`A(4) = ...`	Writable	Invalid	Have exclusive copy

Table 5.3: Access permissions and protocol states

gardless of the protocol state. Such a mechanism could be used to implement the race-detection protocols without the need for keeping blocks invalid.

Currently, there is a Tempest implementation on the CM-5, Blizzard-S [58], that does maintain data on invalid blocks. Blizzard-S separates the notion of protocol access control from the access control maintained by hardware or the operating-system. At the protocol level, blocks are tagged as invalid, read-only, or read-write. Blocks can be cached in a writable state (in hardware) despite being tagged invalid by the protocol. Data values can therefore be maintained on blocks tagged invalid. Table 5.3 shows the updated permissions for the previous example.

Blizzard-S [58] implements fine-grained access control by using a binary-re-writing tool to insert software lookups before loads and stores to shared memory. The lookups examine a table of block tags to determine whether the pending access is allowed. If not, the coherence protocol is directly invoked. This is in contrast to Blizzard-E [58], which leverages fine-grained access control off of operating-system page protections and block-level ECC codes.

```
Process 1:        Process 2:        Process 3:

                  A(1) = 2
                                    A(2) = 3
A(1) = 1
```

Figure 5.3: False race hiding genuine race

Access History Details

The techniques in the previous section ensure the coherence protocol is invoked on all references to shared data. The protocol can be used to monitor memory accesses by updating an access history for referenced locations on each invocation. Like others [33, 46, 47], I choose to keep records of only the most recent read and write to a variable for reasons of efficiency. Races can be missed as a result, if there are multiple races involving the same location, but at least one race involving a location is guaranteed to be caught and can be used to debug parallel programs.

The access histories for each location on a cache block are transferred along with the data, as blocks move from processor to processor. Entries in the history can be as simple as a pair of bits, one each to denote whether a location has been read or written. While bits require less space than byte-entries, they also reduce the amount of information available for describing a race and can cause spurious races to be reported. Byte-entries are an improvement over bits with respect to accuracy of race detection, since they allow the ID of the processor making the most recent read and write to be recorded. I have implemented both approaches and discuss the tradeoffs in Section 5.4.3.

Race detection can be performed at various granularities. For applications that manipulate large shared-memory objects, it might be acceptable to keep access histories for cache-block sized regions of memory or larger. However, this degrades race-detection accuracy if the application actually shares data at finer granularities. Spurious races can be reported in the presence of false sharing, and can hide genuine races. An example of this behavior is shown in Figure 5.3. Assuming the race-detection granularity is large enough to encompass both `A(1)` and `A(2)`, a spurious race between processes 2 and 3 will be reported during the write to `A(2)`. When the second write to `A(1)` is performed, another spurious race will be reported between processes 1 and 3. The access history at this point, holding a single entry, no longer records the first write to `A(1)` and the race between processes 1 and 2 is missed. Thus, unless race detection is performed at a granularity as fine as the sharing in a program, the results cannot be considered accurate. Results for granularities varying from word-level to block-level are reported in Section 5.6.

In Figure 5.3, the race between processes 2 and 3 is discovered when process three writes `A(2)`. We therefore have complete information about the second access of the pair (the *sink*), but all information about the first access (the *source*) must come from the access history. When the protocols detect a race, they report the PC value of the sink, the memory address involved and, when available, the ID of the processor that made the source reference. (The ID is not known unless the access history keeps bytes of information about previous references instead of bits.) The PC of the first access could be maintained as well, but would at least

```
Process 1:          Process 2:

A(1) = 1
                    Z = A(1)
                    A(1) = 2
Z = A(1)
```

Figure 5.4: Monitoring the first read and write

quadruple the size of the access histories. Since histories are transmitted across the network with the block data, their size influences the bandwidth requirements of the protocol. Bandwidth considerations were given priority here over race report detail.

Once the decision to keep incomplete information about the source reference is made, the protocol need not be invoked on *every* access to shared memory. Only the first read and write after fetching a block must be monitored. The protocol still detects all races for which these references are the sinks, and the access-history updates they perform are sufficient to catch races for which these or any later references are the source. Figure 5.4 shows an interleaving of accesses to `A(1)` that produces two races. The read of `A(1)` by process 2 completes a race with the write by process 1, and must therefore be monitored. The write to `A(1)` by process 2 must also be monitored so the access history for the region containing `A(1)` is properly updated. Any subsequent reads or writes by process 2 are redundant (with respect to race detection) since the access history *already* records process 2 as the most recent reader and writer. The read by process 1 at the end of the example is the first after fetching a copy of the block containing

`A(1)`, and must be monitored to detect the race with the write on process 2.

By monitoring only the first read and write after obtaining a copy of a block, we reduce the number of times the protocol must be invoked. This can result in substantial performance gains, as is shown in Section 5.6. Unfortunately, Tempest implementations change access tags at the granularity of an entire block. Thus, we cannot make a block readable, for example, until each race-detection region on the block has been read. Doing otherwise risks missing races since accesses can be missed.

5.4.2 Detecting Concurrency

As with other system-level race-detection schemes [47, 53], coherence protocols are only aware of the processor-level concurrency present in an executing application. This is independent of the means used to express the parallelism (i.e. `fork` calls, `DOALL` loops, Parmacs `CREATE` [15]). For the race-detection protocols, accesses by a pair of processors are concurrent if, during execution, they are not separated by a synchronization event. Synchronization can involve all processors (a barrier), or a subset (locks or joins).

The custom race-detection protocols currently only handle system-wide synchronization, as tests for concurrency in programs using only barriers can be performed efficiently. If access histories are cleared at barriers, non-empty histories imply an access since the last barrier, and testing for concurrency reduces to testing for non-empty access histories. As is seen in Section 5.4.3, testing the access history is implicit in the race-detection test performed at each monitored access, so concurrency tests are essentially free. In the race-detection protocol implementations, processors maintain linked lists of pages containing referenced blocks. The protocol adds pages to the list as part of the access-history updating process. At barriers, processors clear the access histories for blocks on pages in the list and delete the lists.

5.4.3 Detecting Data Races

Section 5.4.1 described techniques to ensure the protocol is invoked on every access to shared memory, and Section 5.4.2 explained how concurrency is detected. This section brings the two components together to find apparent data races on-the-fly. A race-detection protocol has two new responsibilities in addition to its basic role of obtaining copies of data blocks with appropriate access permissions. At each monitored access it must test the access history for references conflicting with the current operation, and it must add the current reference to the update history. The details of when the tests are performed and how the access history is updated determine the exact set of races detected by the protocol.

Figure 5.5 illustrates the new functionality. It shows Teapot code for the protocol handler invoked during a read to a block in an invalid state. (Note that an invalid protocol state is distinct from an invalid block access tag.) As usual, the protocol sends a request for a readable copy of the block to the home node. The `Suspend` statement specifies the protocol state in which to wait for the response. Once the readable copy arrives, the handler makes two new calls. The `TestAccess` routine checks for conflicts between the current access and those recorded in the

```
Message RD_FAULT (id: ID; Var info: INFO; home: NODE)
Begin
    Send(home, GET_RO_REQ, id);
    Suspend(L, SetState(info, Cache_Inv_To_RO{L}));
    TestAccess(info, READ, id);
    SetRdBitsAndLink(info, id);
    Thread_Resume(id);
End;
```

Figure 5.5: Sample race-detection protocol handler

access history. `SetRdBitsAndLink` updates the access history, and ensures that the page containing the block is in the list of modified pages.

Managing Access Histories

The home node maintains a copy of the access history information for each block, and sends it with block data in response to requests from remote processors. As processors manipulate block data, they inspect and update their copies of the history, and eventually return the modified access history to the home when the block is relinquished.

Access histories returned with exclusive copies represent the latest history information in the system, and replace the history data held at the home. If multiple copies of a block have been in circulation, as when there are concurrent readers of a block, the histories must be combined at the home. Protocols using bits to represent access history entires can use a binary OR to merge information about read accesses. A similar operation can be performed for protocols that mark read accesses with processor IDs instead of bits. The race-detection protocols

```
Process 1:          Process 2:

A(1) = 1
                      X = A(1)
                      Y = A(1)
```

Figure 5.6: Multiple races

in this chapter use a unique ID to represent reads by multiple processors. This symbol is left in the access history if more than one of the returning access histories report a location read.

As will be seen below, the technique for combining information about writes depends on the exact scheme used to maintain access histories. Information returning with read-only copies can never contain information recording new writes, but it may be the case that some information about writers is deleted. In the latter case, the home must delete corresponding information from its copy of the access history. Schemes using bits can accomplish this with a binary AND.

Controlling Reported Races

If protocol handlers tested for races on every access, a large number of uninteresting races could be declared. Consider the references shown in Figure 5.6. The first read by process 2 will be discovered to conflict with the write by process 1, as will the second. The second read is unordered with respect to the write, but is less interesting from a debugging point of view since it is strictly ordered with respect to the earlier read. Any races involving the second read must involve the first. Avoiding detection of these additional, uninteresting races is desirable, as it leaves fewer races to be considered by the user. The race-detection protocols already report a subset of the apparent data races in an execution, since they bound access histories. Reporting even a single race from the set involving each location is sufficient for notifying users of program errors, and reporting large numbers of races can be counterproductive as it forces users to examine an increased amount of information.

In general, an access A is the source for a (possibly empty) set S of races. We will say that the set S is *covered* by the race R whose sink is the first access to conflict with A. For example, in Figure 5.6, the write to `A(1)` is the source of races with both reads by process 2. The read of `A(1)` in the assignment to `X` covers the set of races, as it comes first. Our goal is to report only the covering race from each set, when it can be found without unreasonable overhead.

Covering races bear a similarity to Netzer and Miller's first races [50], as they capture the notion of an initial race involving a given memory location. The key difference is that covering races are reported each time a processor acquires a block. Since processors can repeatedly return and reacquire blocks, the set of covering races is much larger than the set of first races. Covering races are therefore less useful from a diagnostic point of view, but help reduce the number of reported race events.

Finding the covering races involves determining an order between the sinks in each set of races. This can be done easily on-the-fly. Accesses are ordered by the program text if they occur on the same processor. Accesses on different processors are ordered by the time at which they request a copy of the accessed

```
Message RD_FAULT (id: ID; Var info: INFO; home: NODE)
Begin
    Send(home, GET_RO_REQ, id);
    Suspend(L, SetState(info, Cache_Inv_To_RO{L}));
    If (not(BeenRead(info, id))) Then
        TestAccess(info, READ, id);
    Endif;
    SetRdBitsAndLink(info, id);
    Thread_Resume(id);
End;
```

Figure 5.7: Guarded race-detection protocol handler

block from the home node. The ordering information can be used to restrict the set of detected races by determining when to call `TestAccess`. If the current processor has already read a location, for example, there is no reason to test subsequent reads as the first read is sufficient to discover the covering race. Figure 5.7 shows a version of the protocol handler in which the access test is guarded by a conditional that only applies the test during the first read.

Unfortunately, guarding the access test does not always limit detected races to covering races. Implementing the `BeenRead` guard requires a history of local accesses be maintained by each processor, as well as the history of accesses made by other processors. `BeenRead` is true if the local history reflects a previous read. Local histories must be cleared each time a block is acquired or the guarding mechanism can cause races to be missed. False sharing at the block granularity can cause non-covering races to be declared by forcing a block to be relinquished and reacquired, clearing the local history in the process. This is illustrated in

```
Process 1:          Process 2:

A(1) = 1
                     Z = A(1)
A(2) = 2
                     Z = A(1)
```

Figure 5.8: Multiple races due to false sharing

Figure 5.8. The write to `A(2)` causes the block to be taken away from process 2 after its first read of `A(1)`. The second read reacquires the block and performs a `TestAccess`, because the local history was cleared when the block arrived. The second read therefore declares a (non-covering) race with the write of `A(1)`.

This example shows that the guarding technique is not always accurate enough to detect just the covering races. The race-detection protocols take an improved approach and alter access histories once races are found. Instead of using guards to limit tests to the access history, `TestAccess` is extended so that, when a race is detected, the source access is removed from the history. The first read in Figure 5.8 now declares a race *and* deletes the record of the write to `A(1)` by process 1. When the second read is performed, `TestAccess` is called but finds no race. No more races will be detected involving `A(1)` until another processor makes an access.

The "history rewriting" approach more accurately limits detection to covering races, but still detects non-covering races when there are concurrent readers following a write. Figure 5.9 shows processes 2 and 3 consuming a value produced by process 1. Process 2 acquires a readable copy of the block from the home when it attempts to read `A(1)`. Along with the block data comes the access history

```
Process 1:       Process 2:       Process 3:

 A(1) = 1
                  X = A(1)
                                   Y = A(1)
```

Figure 5.9: Multiple readers after a write

showing process 1 as a writer. When process 2 calls `TestAccess`, it discovers a conflict with the write, declares a (covering) race, and removes evidence of the write from the access history. But news of this deletion does not reach process 3. The history altered by process 2 remains local until the block is returned to the home. When process three fetches a copy of the block, it contains the original history showing the write by process 1. Thus, both of the readers declare races.

Declaring races at both reads is not unreasonable, since both are unordered with respect to the write and constitute apparent data races. It falls short of the goal of only declaring covering races, but the notion of a covering race was introduced only to reduce, where possible, the number of reported races. Covering races could be found, exclusively, if one were willing to permit a single reader at a time, or implement protocol schemes whereby readers after the first were required to obtain the most recent access history from earlier readers. The small reduction in detected races these techniques offer was not considered valuable enough to offset the performance overheads they would command.

Note that false sharing can now potentially bring us closer to the goal of detecting covering races, instead of causing non-covering races to be declared as it did with the guarding approach. Since false sharing forces processors to prematurely relinquish blocks, it can cause modified access histories to become visible sooner that they would have otherwise. In Figure 5.9, if false sharing had caused process 2 to relinquish its block immediately after modifying the access history in Figure 5.9, it might reached the home in time to be passed to process 3.

Bits Versus Bytes

Either bits or bytes can be used as entries in an access history. Bits have the advantage of requiring less space than bytes, and can be manipulated efficiently, but they are inferior to bytes in several respects. Bytes record the ID of the processor that made an access. This provides additional information about the source reference when races are detected. More importantly, it allows processors to distinguish between their accesses and accesses performed by other processors. Both techniques are described here since byte-schemes could potentially be more expensive to implement on some systems.

Schemes that use bits must keep separate histories for accesses performed locally and remotely. Accesses performed before the block was obtained are recorded in the "past" history, while the "present" history records local accesses. If past and present references were indistinguishable, consecutive accesses by a single processor could mistakenly be declared a race. Schemes that use bytes to implement access histories can easily make this distinction without the separate histories.

In the bit-history schemes, past and present histories are combined and returned with the data when a block is surrendered. Thus, once a block is relin-

```
Process 1:          Process 2:

A(1) = 1
                    A(2) = 2
X = A(1)
```

Figure 5.10: Spurious race caused by false sharing

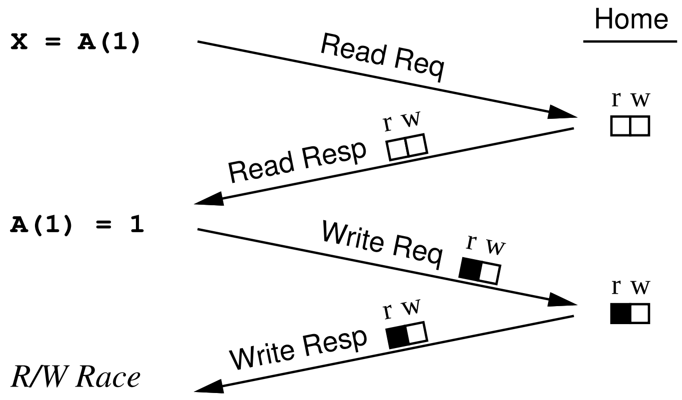


Figure 5.11: Example of a self race

quished and reacquired, processors cannot distinguish their accesses from others in the history. Figure 5.10 shows a spurious "self race" being declared as a result of false sharing. Process 1 writes `A(1)` and sets the write bit in the present access history. Process 2 acquires the block containing both `A(1)` and `A(2)`. When process 1 reacquires the block to read `A(1)`, it finds the write bit set in the past history and declares a race — even though the write was actually performed by process 1.

As shown in Figure 5.11, self races can arise even without false sharing when a processor reads and writes a location. At the read of `A(1)`, the processor acquires a read-only copy of the block and sets a bit in the present history. When the processor tries to write the location, a request for write permission must be sent to the home. To ensure that races elsewhere in the system are not missed, the write request must carry a copy of the access history bits. When the response to the write request arrives, it contains a set read bit, but the processor cannot tell whether it was set locally or by a remote processor, and a race is declared.

Schemes that use bytes to record access-history entries have no problems with self races, since they can distinguish their contributions from those of other processors even after a block has been relinquished and reacquired due to false sharing or other mechanisms. Both approaches are evaluated in Section 5.6, since the bit-history schemes consume less memory, and require less information be sent across the network with block data.

5.4.4 Detected Races

The race-detection protocols find apparent races instead of actual, since their access histories allow them to detect accesses that *could* have overlapped as well as those that actually did. That the protocols find apparent races can be shown using the single-access execution view ([48]), where the events of interest are individual accesses to shared memory. Without access histories, protocols would be limited to detecting actual data races involving events (accesses) that executed concurrently. The histories allow detection of conflicting events that are unordered by barrier synchronization. It is assumed that, because of the lack of synchronization, the events could have occurred simultaneously and therefore represent a race. Since this ignores data dependences that could potentially order the events, the detected races are apparent and not necessarily feasible.

As with all approaches that bound access histories [33, 46, 47], the protocols detect a *subset* of the apparent races present in an execution. Races can be missed if there are multiple races involving the same location, but at least one race involving a location is guaranteed to be caught and can be used to debug parallel programs. Finding a subset of the apparent races is not necessarily a handicap. Experience with the race-detection protocols has shown that reducing the number of reported races makes the protocols more useful as diagnostic tools, since users need examine less information, and reporting even a single race for a location is sufficient to warn the user of an error. The effort to report covering races (Section 5.4.3) is an attempt to further limit the number of races reported.

It is not possible to precisely describe the subset of the apparent races that will be detected, since the protocols use system-level information to detect races, and an application's system-level behavior can vary from run to run. The races detected depend upon the ordering of accesses to a given location, and can therefore change from run to run. This is not problematic, as the protocols are still guaranteed to find at least one of the set of races involving a location — including a first race [50] for each location.

A larger concern is the selection of race-detection granularity. As Section 5.4.1 shows, race-detection accuracy is compromised when the granularity at which races are detected is larger than that at which data is shared in an application. To be safe, detection should therefore be performed at the word-level unless users are sure that larger granularities are appropriate.

Finally, the custom race-detection protocols currently only recognize system-wide synchronization (barriers). This limitation is not as severe as it might first appear. Atomicity of critical sections can often be ensured statically, without any need for runtime monitoring. Four of the five benchmarks used in Section 5.6 are amenable to this approach, as the pairwise synchronization operations are clearly identifiable, and there are no problems with aliasing or indirection that would dilute the precision of static analysis. Statically analyzing the remaining benchmark would require interprocedural analysis, as it calls functions from within critical sections.

Applications with additional forms of synchronization can be run on the protocols, but spurious races may be declared since only the ordering constraints imposed by barriers are recognized by the race-detection mechanisms. As is shown in Section 5.6.3, these spurious races can be effectively removed in a post-processing phase.

5.5 Verification

The formal verification tool associated with Teapot [22, 24] was used to ensure both that the race-detection protocols maintained consistent data, and that they successfully caught data races. The verification process proceeded in two steps. First, the basic protocol was designed and tested without any race-detection functionality. Even without race-detection features, the protocol is significantly more complex than the Stache protocol, since it must keep blocks in an invalid state until each location in a block has been read or written. Once the underlying protocol was working correctly, mechanisms for recording access histories and detecting races were added. The details of the race-detection functionality left opportunities for error, so efforts were made to verify the protocol correctly caught apparent data races as well.

The verification process exhaustively explores a state space, the size of which is related to the complexity of the protocol and system configuration being verified. This made the protocols that use bits for access-history entries more attractive than those that use bytes. A verification state for bit schemes can be encoded in less space, allowing more states to be explored in a given amount of memory. Also, fewer states are required to exhaustively verify the protocol since its access histories contain less information than the byte schemes. (Access histories in bit schemes hold a smaller number of distinct entries than do byte schemes, so fewer states need be explored.) All verification attempts were therefore on a bit-history protocol, with one location per block. Differences between the bit-history and byte-history protocols are confined to the support routines that manage access histories — protocol states and transitions are identical. Thus, ensuring that the bit-history protocols function correctly gives increased confidence that the byte scheme works as well.

Informally, verification exhaustively tests a protocol by simulating a stream of loads and stores to shared memory. Since the verification system has perfect knowledge of the accesses performed, it can be extended to recognize data races along with the protocol. The system was modified to maintain access histories for

Processors	*Locations*	*Reordering*	*States*	*Rules*	*Seconds*
2	1	0	3,866	10,288	11.9
2	1	1	10,084	25,948	31.6
2	1	2	17,448	50,068	68.4
2	*2*	*0*	*947,502*	*2,978,195*	*3,737.8*
2	*2*	*1*	*947,509*	*2,535,105*	*3,285.1*
2	*2*	*2*	*852,751*	*2,190,663*	*2,800.8*
3	*1*	*0*	*778,662*	*1,944,167*	*3,506.9*

Table 5.4: Protocol verification results

each memory location, and check these histories for conflicting accesses at each simulated reference. Whenever the verification system found a race, it tested the protocol to see that it also detected the race. This guaranteed that the protocol caught all apparent data races that could be found with an access history of size one, but left open the possibility that it might generate spurious race reports as well. Since bit schemes *do* generate spurious race reports, this is exactly the behavior required.

Table 5.4 presents the results of the verification runs. The first three columns describe the tested configuration. Systems with two or three processors, one or two memory locations, and varying amounts of network reordering were considered. The remaining columns list the size of the verification state space, the number of rules the system fired in exploring the state space, and the verification time in seconds. Entries in italics are for runs that could not exhaustively explore the required state space in 150MB of memory, and therefore did not complete. Only configurations with two processors and one memory location could be exhaustively

Application	*Description*	*# Lines*	*Synchronization*
Appbt	3D CFD Solver	5,100	Locks, Barriers
Em3d	EM Wave Propagation	2,500	Two locks, Barriers
Gauss	Gaussian Elimination	900	One lock, Barriers
LCP	Linear Complementarity	1,550	One lock, Barriers
Water	Molecular Simulation	2,400	Locks, Barriers

Table 5.5: Benchmark applications

verified. While this is insufficient to guarantee that the protocol works correctly on larger configurations, it greatly increases confidence that it will do so. Verifying as much as possible of other configurations adds to this confidence, as it ensures that there are no errors in the explored space.

5.6 Performance

This section describes the performance of the custom race-detection protocols on a set of five benchmarks. Schemes that use bytes to record access-history entries are shown to produce fewer spurious races, in general, than bit-schemes, though the difference on races missed is much smaller. Slow-downs range from zero to less than a factor of three. Actual program errors were found in two of the benchmarks, Appbt and LCP, by the custom race-detection protocols.

5.6.1 Benchmarks

Each of the benchmarks used to test the race-detection protocols (Table 5.5) use both locks and barriers for synchronization. Since the race-detection protocols only recognize barrier synchronization, critical sections implemented by locks appear to consistently fail from a race-detection point of view. This causes detection of spurious races involving references in critical sections, and these were mechanically separated from the others when trying to determine how useful and effective the protocols are at finding races in programs with barrier synchronization.

Appbt is a kernel from the NAS parallel benchmarks representing the computation and communication found in 3D computational fluid dynamics problems. It makes frequent use of both locks and barriers for synchronization. Em3d models the propagation of electromagnetic waves through objects in three dimensions. The simulation is formulated as a computation on a bipartite graph with directed edges, and uses only barrier synchronization during the computation phase. The graph-building phase uses two locks. Gauss performs gaussian elimination and backsubstitution. It uses a single lock to implement a reduction across processors, and barrier synchronization elsewhere. LCP is a parallel implementation of the linear complementarity problem, written by Satish Chandra. Only barrier synchronization is used during the computation, though a single lock is used to combine normalization information at the end. Water is one of the Splash [60] benchmarks, and simulates a body of water molecules. The version used here is the original n^2 application, and uses both locks and barriers during the computation.

5.6.2 Experimental Setup

All experiments were performed on a 32-processor CM-5 using the Blizzard-S [58] implementation of Tempest [55]. The baseline against which the race-detection protocols were compared was the Teapot-generated version of the Stache protocol. All protocols maintained coherence at the 32-byte block granularity, and race-detection granularities varied from 4 to 32 bytes. Schemes that used both bits and bytes to represent access history information were tested.

The race-detection protocols will be referred to by names of the form Race-*Type*-*Gran*, where *Type* is either "Bit" or "Byte" and describes the scheme used to record accesses histories, and *Gran* is the race-detection granularity. For example, Race-Byte-4 uses the byte scheme, and detects races at the four-byte granularity.

Five runs of each application were made on each protocol. Normal run-to-run differences in the ordering of accesses to shared memory influenced the races exhibited, so the total number of detected races varied slightly from run to run. Of the five runs, the one reporting the largest total number of races was selected in each case. In the case of a tie, the run completing in the shortest amount of time was chosen.

5.6.3 Race Detection Results

Table 5.6 summarizes the races found by Race-Byte-4, the most accurate of the race-detection protocols. The number of race events is the total number of apparent data races detected. Since the same pair of source-line references can be involved repeatedly in races with each other, more useful metrics are the unique

Application	*Race Events*	*Unique Addresses*	*Unique References*	*Unique Lines*	*Excluding Crit. Sec.*
Appbt	723,333	43,069	163	91	1
Em3d	149,127	43,034	13	3	0
Gauss	26,167	1	2	2	0
LCP	1,078,289	4,161	13	9	2
Water	2,466,122	4,616	36	25	1

Table 5.6: Races detected by Race-Byte-4

memory addresses and references involved in races. After an application runs to completion, the protocol generates a report of detected races, containing the unique addresses and references, in addition to the total number of race events. I have written tools that annotate source code on the basis of this information, and highlight lines involved in races. The number of lines is smaller than the number of references, since each line can contain multiple references. The table also shows the number of source lines remaining after discarding those in critical sections. The remaining line in Appbt represented an actual program error, as did the two in LCP. In neither case were the authors aware of the missing synchronization. The line in Water was the result of a synchronization scheme (spinning on a shared memory value) that was not based on locks or barriers.

Pseudocode summarizing the program error found in Appbt is shown in Figure 5.12. An identical copy of the loop shown is run on every processor in the system. During each iteration, an array is updated and a normalization calculation is performed. The update is performed in parallel, but the normalization, which depends on results from the update phase, is all done on a single processor.

```
FOREACH timestep DO {
    Barrier();
    Update_Array();
    if (Processor_ID == 0) {
        Normalize_And_Print();
    }
}
```

Figure 5.12: Pseudocode showing program error

Without a barrier between these two steps, the update and normalization phases can (and do) overlap. In this case, the final answer is not affected, but the output generated during the normalization phase is unpredictable. Placing a barrier between the update and normalization phases solved the problem. The race in LCP was similar, and also involved the overlap of computation and normalization.

Differences in race-detection granularities and schemes used to maintain access histories affect the accuracy of race detection. Figure 5.13 shows spurious races detected by each of the race-detection protocols. The data was generated by assuming that, for all benchmarks, Race-Byte-4 correctly identified the unique references involved in races. Any additional references were considered false. The graphs show spurious races as a percentage of the total number of non-spurious races. As expected, the approaches that use bit-history schemes can be seen to report many more false races than the byte-history schemes do, even at the finest granularities.

Spurious races are a nuisance, but are not as serious as races that are *missed*. Figure 5.14 shows the missed races for each benchmark, again assuming that Race-

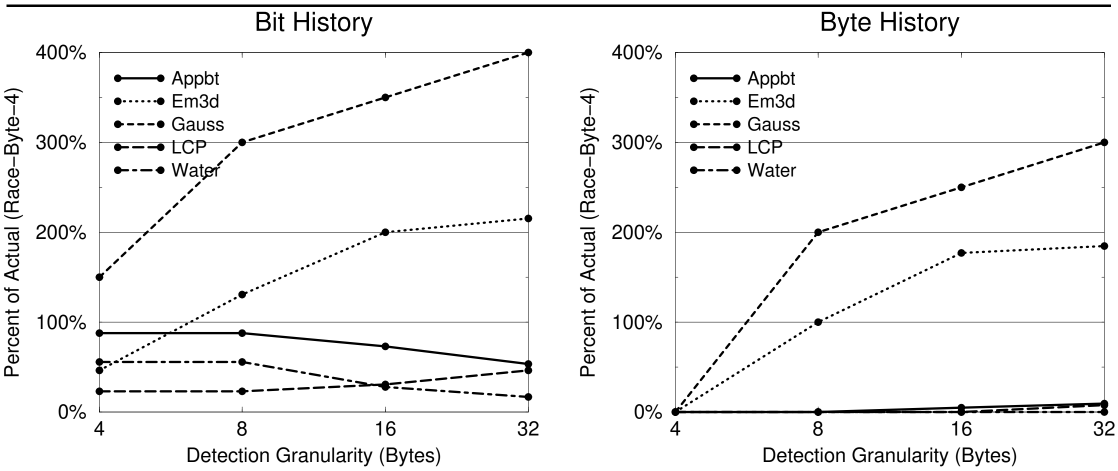


Figure 5.13: Spurious races reported

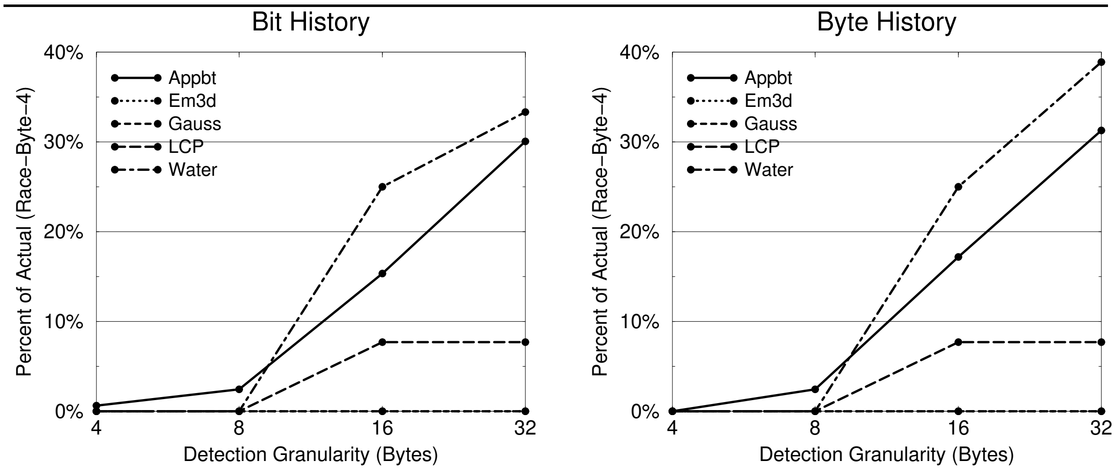


Figure 5.14: Races missed

Byte-4 finds all apparent races. The results show an increase in missed races as the granularity increases, as expected. There is little difference between the the bit-history and byte-history approaches. This is not surprising, as the bit-history schemes should, in general, catch at least as many races as the byte-history.

Summary

The results validated the protocol-based approach to race detection. The protocol scheme found actual program errors in two of the five benchmarks, despite the presence of pairwise synchronization. As expected, byte-history schemes reported fewer spurious races than the bit-history schemes, and increasing granularities caused races to be missed.

5.6.4 Performance Results

Figures 5.15 and 5.16 show the slow-downs for the five benchmarks on each of the custom protocols. Slow-downs range from almost a factor of three on Gauss, to slight *improvements* in Appbt and LCP. The lessons learned during the performance tuning of the LCM protocols (Chapter 3) apply here as well. The performance differences between benchmarks are almost entirely tied to network contention on the CM-5.

A convenient way to assess contention is to measure the average number of attempts required to inject a message into the network. The data on the number of tries per send is shown in Figure 5.7. Both of the applications that see speedups have extremely high levels of contention when run on the Stache protocol, but see

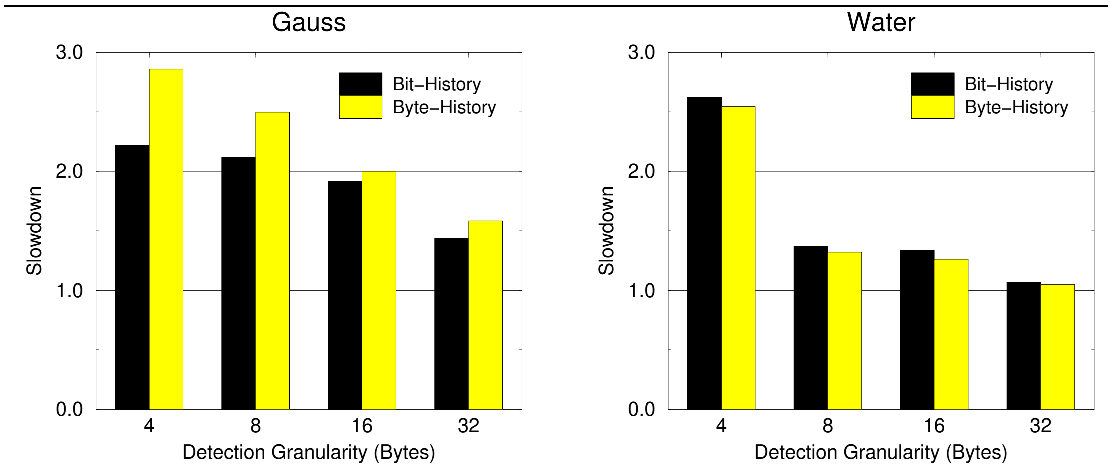


Figure 5.15: Slow-downs for Gauss and Water

Application	*Stache*	*Byte-4*	*Byte-8*	*Byte-16*	*Byte-32*
Appbt	65.7	41.6	54.5	60.7	63.0
Em3d	1.3	1.3	1.1	1.1	1.1
Gauss	1.4	1.8	1.7	1.5	1.3
LCP	13.6	7.1	6.9	6.5	8.1
Water	1.3	2.0	1.2	1.2	1.3

Table 5.7: Network contention (tries per send)

lower levels of contention when race-detection protocols are used.

In general, as race-detection granularities decrease, protocols are less likely to access each race-detection region on a block. This implies an increase in the number of faulting accesses, since fewer blocks can be upgraded to read-only or writable states. The bulk of the extra faults can be handled locally, without sending or receiving data, and so execute quickly. At each fault, the Tempest system pulls waiting messages out of the network and queues them for delivery.

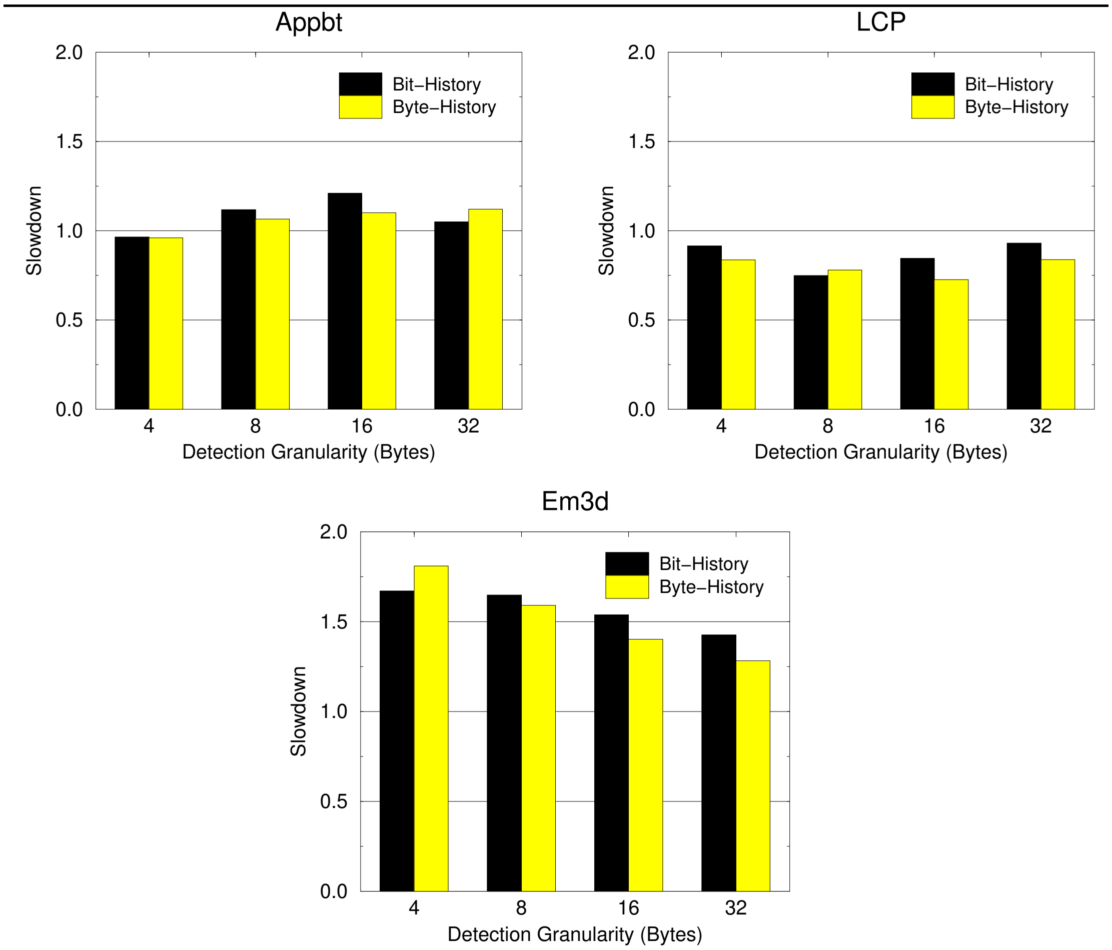


Figure 5.16: Slow-downs for Appbt, LCP, and Em3d

Application	*Clearing Histories*	*Access Tests*	*Total*
Appbt	0.15%	0.68%	0.83%
Em3d	1.10%	0.29%	1.39%
Gauss	7.56%	0.46%	8.02%
LCP	0.38%	0.51%	0.89%
Water	0.16%	0.76%	0.92%

Table 5.8: Race-detection overheads

Thus, faulting more often can be beneficial as it takes pressure off of the network buffers and lowers contention. This mechanism is behind the decreased contention numbers for Appbt and LCP.

But the race-detection protocols transfer more data than Stache does, since they must include access histories. This can exacerbate contention in some situations, as with Gauss and Water. Since these two benchmarks have low contention initially, increasing the number of access faults does not help. Instead, the increase in network traffic drives up contention and gives sizeable slow-downs on the race-detection protocols.

In Water, there is a large discontinuity between the slow-downs for four and eight bytes. This is because all data in Water is double-precision. When race detection is performed at the four-byte granularity, the protocol assumes that all accesses (even of eight-byte quantities) touch four bytes at a time. Since there are always unreferenced four-byte quantities on each block, blocks can never be upgraded to read-only or writable and the protocol must fault on every reference.

Network contention is the dominant factor in determining how well an ap-

App.	*Total Refs (M)* *Read*	*Write*	*Stache* *Read*	*Write*	*Race-Byte-4* *Read*	*Write*	*Dilation* *Read*	*Write*
Appbt	471	16	0.1%	2.7%	58.9%	100.0%	402	38
Em3d	3	9	33.7%	11.0%	100.0%	100.0%	3	9
Gauss	89	47	0.7%	0.2%	56.1%	99.7%	83	574
LCP	138	1	1.0%	16.3%	74.1%	100.0%	76	6
Water	135	16	0.7%	4.2%	98.7%	100.0%	141	24

Table 5.9: Statistics on read and write faults

plication will fare on the data-race protocols on the CM-5. The per-processor overheads introduced by race detection, shown in Table 5.8, are all quite small. The additional cycles spent clearing access histories at barriers and performing tests for races account for less than *1%* of the total execution time for three of the five benchmarks. This gives reason to hope that the largest slow-downs will decrease on systems with greater bandwidth, such as networks of workstations.

Table 5.9 reports data on the fraction of the accesses made by each application that fault and invoke the protocol. The total number of accesses were measured with a version of Race-Byte-4 that instrumented all accesses. This was used as a baseline against which to compare the number of faulting accesses for Stache and Race-Byte-4. The final columns report the relative increase in monitored accesses when using Race-Byte-4. The data indicates that the optimization of upgrading a block to read-only or writable once each region has been accessed only slightly decreases the number of faulting accesses when using Race-Byte-4.

Summary

The performance results were surprising in several respects. First, program *speed-ups* were not anticipated. It should be stressed, however, that they were the result of system-dependent factors (i.e. the shallow network on the CM-5). Also, the performance of the bit-history and byte-history schemes was nearly equivalent. It was expected that the additional bandwidth required to support the byte-history approach would result in larger slow-downs than seen with bit histories. Finally, while the number of access faults was expected to increase when using the race-detection protocols, the magnitude of the increase in some cases was surprising. The relatively minor slow-downs are impressive given the dilations in Table 5.9.

5.7 Conclusions

This chapter has described the design, verification, and implementation of a family of custom cache-coherence protocols that perform on-the-fly detection of apparent data races for programs with barrier synchronization. Efficient detection of data races is possible on DSM systems because a mechanism is already in place to invoke the coherence protocol in response to shared-memory accesses. The protocol can be extended to maintain access histories, detect concurrency, and watch for data races. Protocol-based race detection schemes are completely independent of program source code, and race detection can be performed on programs written in any language and on library routines for which the source may not be available.

Overheads for these protocols range from zero to less than a factor of three, though there is reason to believe that this performance will improve on systems with greater bandwidth. In comparison, Perković and Keleher [53], the only other system-level race-detection implementation, report slow-downs ranging from five to 30. The on-the-fly method of Dinning and Schonberg [25] has slow-downs ranging from three to six, though they handle a larger class of synchronization. Hood, Kennedy, and Mellor-Crummey [33] have lower overheads than the protocol-based techniques, at approximately 40%, but require compiler involvement.

Chapter 6

Conclusions

Distributed Shared-Memory (*DSM*) computers, which partition physical memory among a collection of workstation-like computing nodes, are emerging as the way to implement parallel computers, as they promise scalability and high performance. Shared-memory DSM machines use a coherence protocol to manage the replication of data and to ensure that a parallel program sees a consistent view of memory. A protocol determines, to a large extent, the performance of a shared-memory program since communication occurs through loads and stores to shared data.

Applications have very different patterns of communication and no single, general-purpose protocol suits all programs. This has prompted interest in systems that enable users to select a coherence protocol and, more recently, in systems in which the protocol is implemented in software instead of being fixed in hardware. DSM machines with software-implemented coherence protocols provide opportunities for a variety of more complex and application-specific protocols and *allow for protocols that do not just ensure consistent memory, but also provide new functionality and semantics.*

Parallel programming has long faced a tension between the goals of high performance and ease of use. Languages and tools can make parallel computers easier to use, but concerns about their efficiency have limited their usage. This thesis demonstrates that some high-level languages and tools can be implemented more efficiently by taking advantage of the cache coherence protocols that underly software DSM machines, thereby improving both performance *and* ease of use.

6.1 Thesis Summary

As proof that protocol support can improve the performance of parallel language implementations, this thesis has presented the design, implementation, and verification of a family of custom protocols that efficiently support a large-grain data-parallel language C**. In C**, to prevent data access conflicts, processors' modifications of memory must be kept local until all parallel tasks complete. Loosely Coherent Memory systems transparently copy modified locations at the protocol level, and efficiently reconcile these copies at the end of each parallel phase. On programs for which static analysis is imprecise, LCM support improves performance from a few percent up to a factor of 3, and reduces memory overheads from a factor of 2 to a factor of 5 over a compiler-copying scheme.

LCM memory systems are more generally useful as well. LCM's update mechanism and its ability to reconcile multiple copies of data can significantly reduce overheads associated with fine-grained sharing of data. On a set of four applications written in C, LCM support improved overall performance by up to a factor of 3, and increased the performance of selected program phases by as much as a factor of 4.7.

Finally, this thesis described the design and implementation of custom cache-coherence protocols that perform on-the-fly detection of actual data races for programs with barrier synchronization. Efficient detection of data races is possible on DSM systems because a mechanism is already in place to invoke the coherence protocol in response to shared-memory accesses. The protocol was extended to maintain access histories, detect concurrency, and watch for data races. Overheads in execution time for the race-detection protocols range from zero to less than a factor of three — a significant improvement over comparable approaches. Race-detection protocols found actual program errors in two applications.

6.2 Future Work

The work in this thesis can be extended in a number of ways. First, LCM could be modified to support C** programs containing nested parallelism. With nested parallelism, modifications are hidden from other invocations, but must remain visible to an invocation's child processes. Read requests must therefore be satisfied by the parent process, and not necessarily the data's home. As nesting levels increase, read requests may have to be propagated up the ancestor tree in search of the correct value.

Also, LCM performance could potentially be improved by optimizing for the case where only one modified copy of a given block is created. Currently, there is no way to know how many processors will modify a block, so an accumulator copy is created and initialized when the home receives the first modified block. This overhead could be eliminated by delaying the creation of an accumulator copy until a second modified copy arrived. This could most easily be done with LCM-MCC, since modified blocks are already flushed in bulk and buffered. It may be possible to sort buffered blocks or otherwise determine which modified copies are unique.

Applications like LCP mark large regions of data before performing modifications. Currently, memory blocks are requested individually, and the mark directive does not return control to the processor until all blocks in the region have been acquired. In these cases, the mark could instead be treated as a prefetch. Processors could be allowed to proceed immediately, and would only delay if they attempted to access a block that had not yet arrived. This scheme has the advantage of allowing the prefetched data to be sent by the home in bulk as well.

Chapter 4 showed that LCM support could improve performance in languages other than C**, but the improvements were obtained by manually inserting memory-system directives into applications. A tool for automatically inserting these directives would ease the burden of correctly augmenting applications. Points at which applications acquire a lock before modifying a value suggest that LCM support is potentially useful, as it can efficiently combine modifications and eliminate locks. Determining how and where to use the update facility is more difficult, and requires that communications patterns be statically analyzable.

Finally, the race detection protocols would be much more useful if they were extended to handle pairwise synchronization. Barrier synchronization is easier to handle, since accesses by a processor between barriers are unordered with respect to *all* other processors in the system. With pairwise synchronization, tests for races are more involved since prior accesses to a location may or may not be races depending upon the pairwise orderings imposed by synchronization. If variables are associated with locks, access histories could be extended to contain information about locks held by a processor during an access. Races are caused by processors referencing locations without first acquiring the necessary lock. Also, the more general vector time-stamp [37] technique could be used to detect orderings imposed by pairwise synchronization, but at the cost of an increase in the amount of data transmitted with each block.

Bibliography

[1] Sarita V. Adve and Mark D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, 1993.

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proc. of the 22th Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 2–13, June 1995.

[3] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. of the 15th Annual Int'l Symp. on Computer Architecture (ISCA'88)*, pages 280–289, 1988.

[4] Randy Allen, Donn Baumgartner, Ken Kennedy, and Allan Porterfield. PTOOL: A Semi-Automatic Parallel Programming Assistant. Technical Report TR86-31, Rice University, Department of Computer Science, January 1986.

[5] Todd R. Allen and David A. Padua. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, University Park PA, August 1987.

[6] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwanepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[7] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the SIGPLAN '88 Confer-*

ence on Programming Language Design and Implementation (PLDI), pages 11–20, June 1988.

[8] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991.

[9] William F. Appelbe and Charles E. McDowell. Anomaly Reporting: A Tool for Debugging and Developing Parallel Numerical Algorithms. In *Proceedings of the First International Conference on Supercomputing Systems*, pages 386–391, 1985.

[10] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[11] Sandra Johnson Baylor, Kevin P. McAuliffe, and Bharat Deep Rathi. An Evaluation of Cache Coherence Protocols for MIN-Based Multiprocessors. In *International Symposium on Shared Memory Multiprocessing*, pages 230–241, Tokyo, April 1991.

[12] Monica Beltrametti, Kenneth Bobey, and John R. Zorbas. The Control Mechanism for the Myrias Parallel Computer System. *Computer Architecture News*, 16(4):21–30, September 1988.

[13] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 168–176, February 1990.

[14] B. N. Bershad and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, September 1991.

[15] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfieldand Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston Inc., 1987.

[16] David Callahan and Jaspal Subhlok. Static Analysis of Low-Level Synchronization. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel*

and Distributed Debugging, published in ACM SIGPLAN Notices, 24(1):100–111, January 1989.

[17] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[18] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[19] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[20] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. Technical Report CSL-TR-93-597, Department of Computer Science, Stanford University, December 1993.

[21] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 61–75, October 1994.

[22] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language Support for Writing Memory Coherence Protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[23] Jong-Deok Choi, Barton P. Miller, and Robert Netzer. Techniques for Debugging Parallel Programs with Flowback Analysis. Technical Report 786, University of Wisconsin, Madison, Computer Sciences Department, August 1988.

[24] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[25] Anne Dinning and Edith Schonberg. An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, February 1990.

[26] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Z waenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, pages 144–155, May 1993.

[27] Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):89–99, January 1989.

[28] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[29] Michael J. Feeley and Henry M. Levy. Distributed Shared Memory with Versioned Objects. In *OOPSLA '92: Seventh Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 247–262, October 1992.

[30] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, June 1990.

[31] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0, May 1993.

[32] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*.

[33] Robert Hood, Ken Kennedy, and John Mellor-Crummey. Parallel Program Debugging with On-the-fly Anomaly Detection. Technical Report TR90-111, Rice University, Department of Computer Science, May 1990.

[34] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[35] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical Report 93-214, Department of Computer Science, Rice University, November 1993.

[36] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 302–313, April 1994.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[38] J. R. Larus, B. Richards, and G. Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proc. of the Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, October 1994.

[39] James R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages And Compilers for Parallel Computing (5th International Workshop)*, pages 326–341. Springer-Verlag, August 1993.

[40] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[41] James R. Larus, Brad Richards, and Guhan Viswanathan. Parallel Programming in C**: A Large-Grain Data-Parallel Programming Language. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 8, pages 297–342. MITP, 1996.

[42] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 148–159, June 1990.

[43] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[44] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[45] Tom Lovett and Russell Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, May 1996.

[46] John M. Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing '91*, pages 24–33, November 1991.

[47] Sang Lyul Min and Jong-Deok Choi. An Efficient Cache-based Access Anomaly Detection Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Santa Clara, California, April 1994.

[48] Robert H. B. Netzer. *Race Condition Detection for Debugging Shared-Memory Parallel Programs.* PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, August 1991.

[49] Robert H. B. Netzer and Barton P. Miller. Detecting Data Races in Parallel Program Executions. Technical Report TR90-894, University of Wisconsin, Madison, Department of Computer Science, August 1990.

[50] Robert H. B. Netzer and Barton P. Miller. Improving the Accuracy of Data Race Detection. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP, published in ACM SIGPLAN NOTICES*, 26(7):133–144, July 1991.

[51] Robert H. B. Netzer and Barton P. Miller. What are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1:74–88, March 1992.

[52] *I. Nudler and L. Rudolph. Tools for the Efficient Development of Efficient Parallel Programs. In *Proceedings of the First Israeli Conference on Computer Systems Engineering*, May 1986.

[53] Dejan Perković and Pete Keleher. Data Race Detection in Release-Consistent DSM. In *Operating System Design and Implementation*, 1996. To appear.

[54] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[55] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 325–337, April 1994.

[56] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proc. of the 23th Annual Int'l Symp. on Computer Architecture (ISCA'96)*, May 1996.

[57] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[58] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–307, October 1994.

[59] Edith Schonberg. On-the-Fly Detection of Access Anomalies. *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, published in ACM SIGPLAN Notices*, 24(7):285–297, July 1989.

[60] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[61] Guy L. Steele Jr. Making Asynchronous Parallelism Safe for the World. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 218–231, January 1990.

[62] Richard N. Taylor and Leon J. Osterweil. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–278, May 1980.

[63] Charles P. Thacker and Lawrence C. Stewart. Firefly: a Multiprocessor Workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 164–172, October 1987.

[64] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proc. of the 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, pages 156–168, May 1993. Also appeared in *CMG Transactions,* Spring 1994.

[65] David A. Wood, Garth G. Gibson, and Randy H. Katz. Verifying a Multiprocessor Cache Controller Using Random Case Generation. *IEEE Design and Test of Computers*, 7(4):13–25, August 1990.

Appendix A

The LCM Protocol

This appendix contains the finite-state machine (FSM) descriptions of the LCM protocols. Protocol actions are different on the home and remote processors for a given memory block, and diagrams are presented for each. These diagrams have been automatically generated from the 2,000-line Teapot protocol specification describing the SCC, MCC, SCC-Update, and MCC-Update versions.

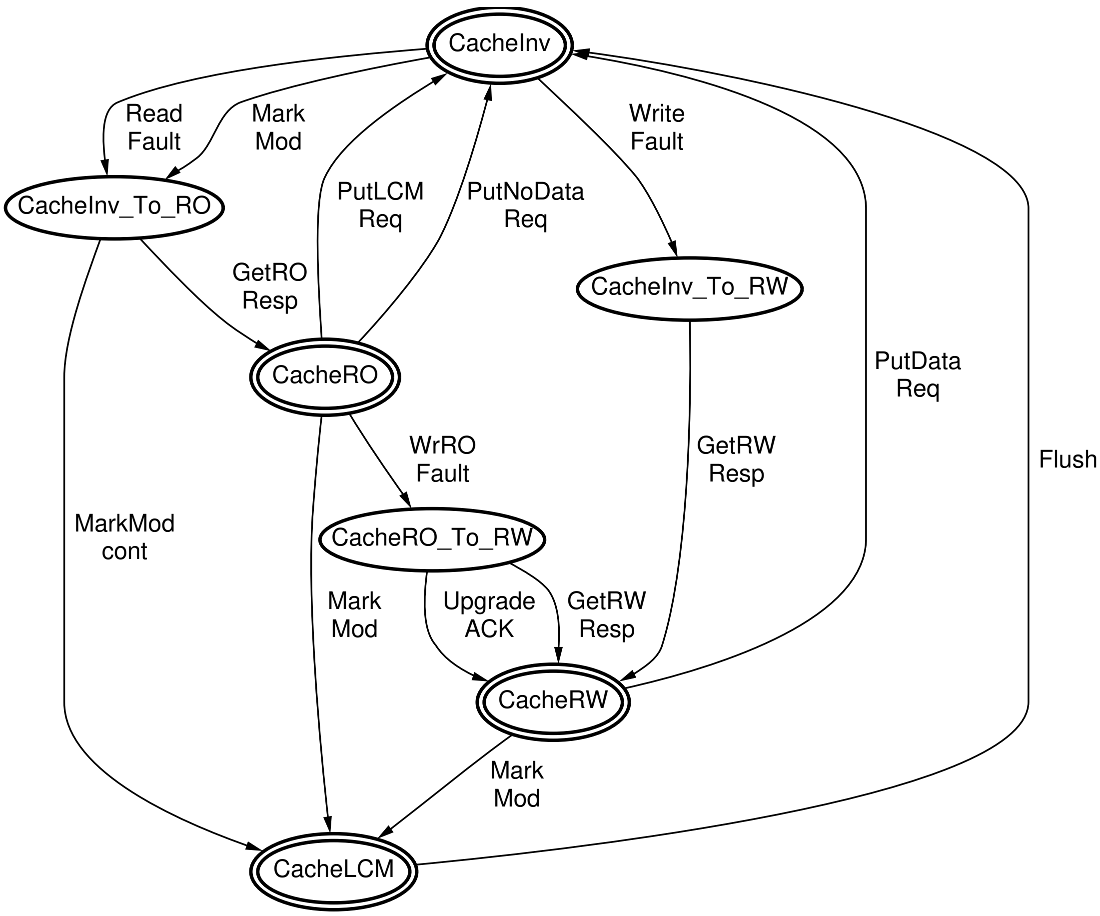


Figure A.1: LCM-SCC remote-side FSM

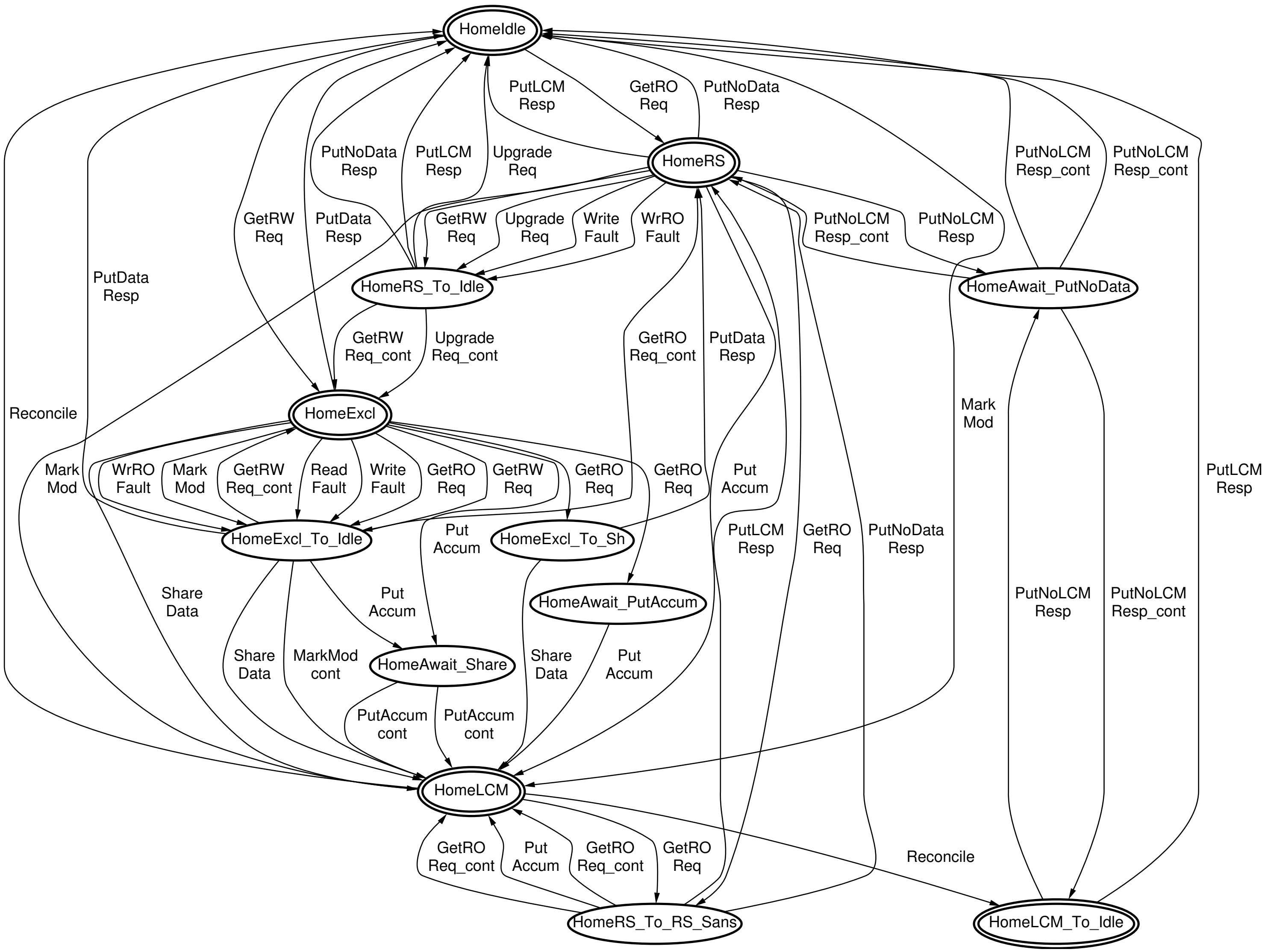


Figure A.2: LCM-SCC home-side FSM

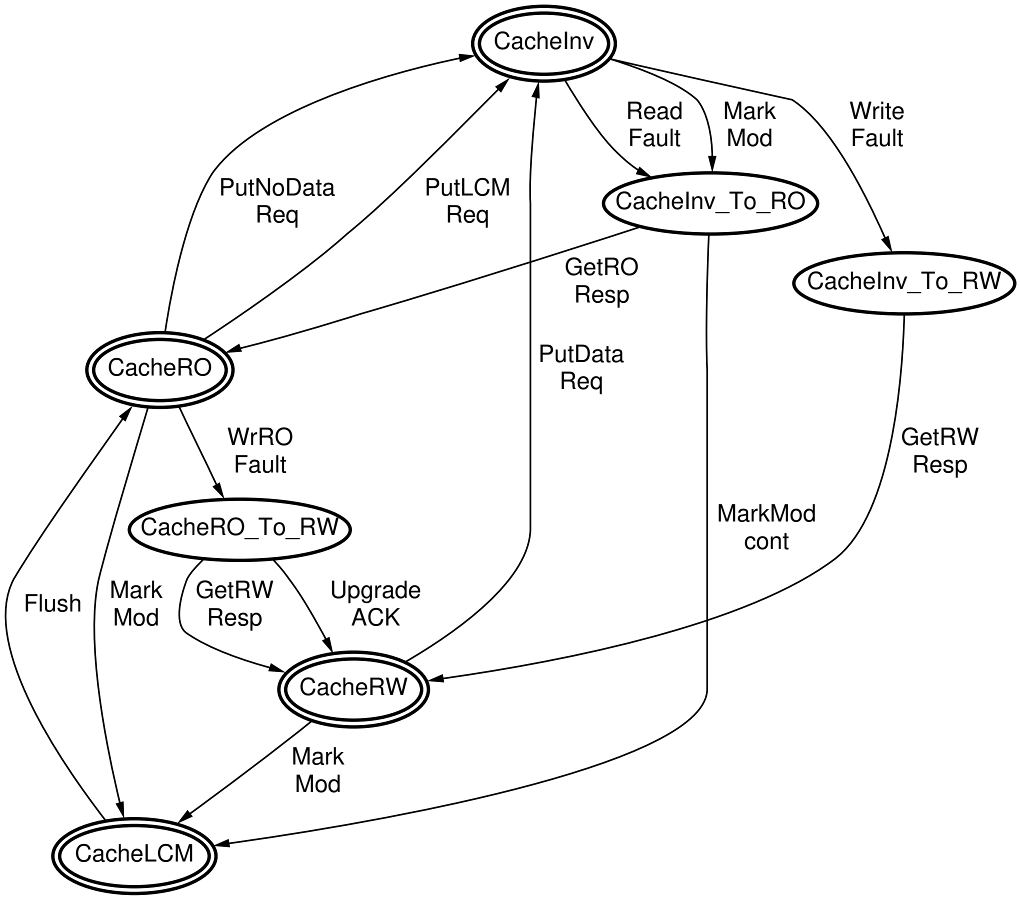


Figure A.3: LCM-MCC remote-side FSM

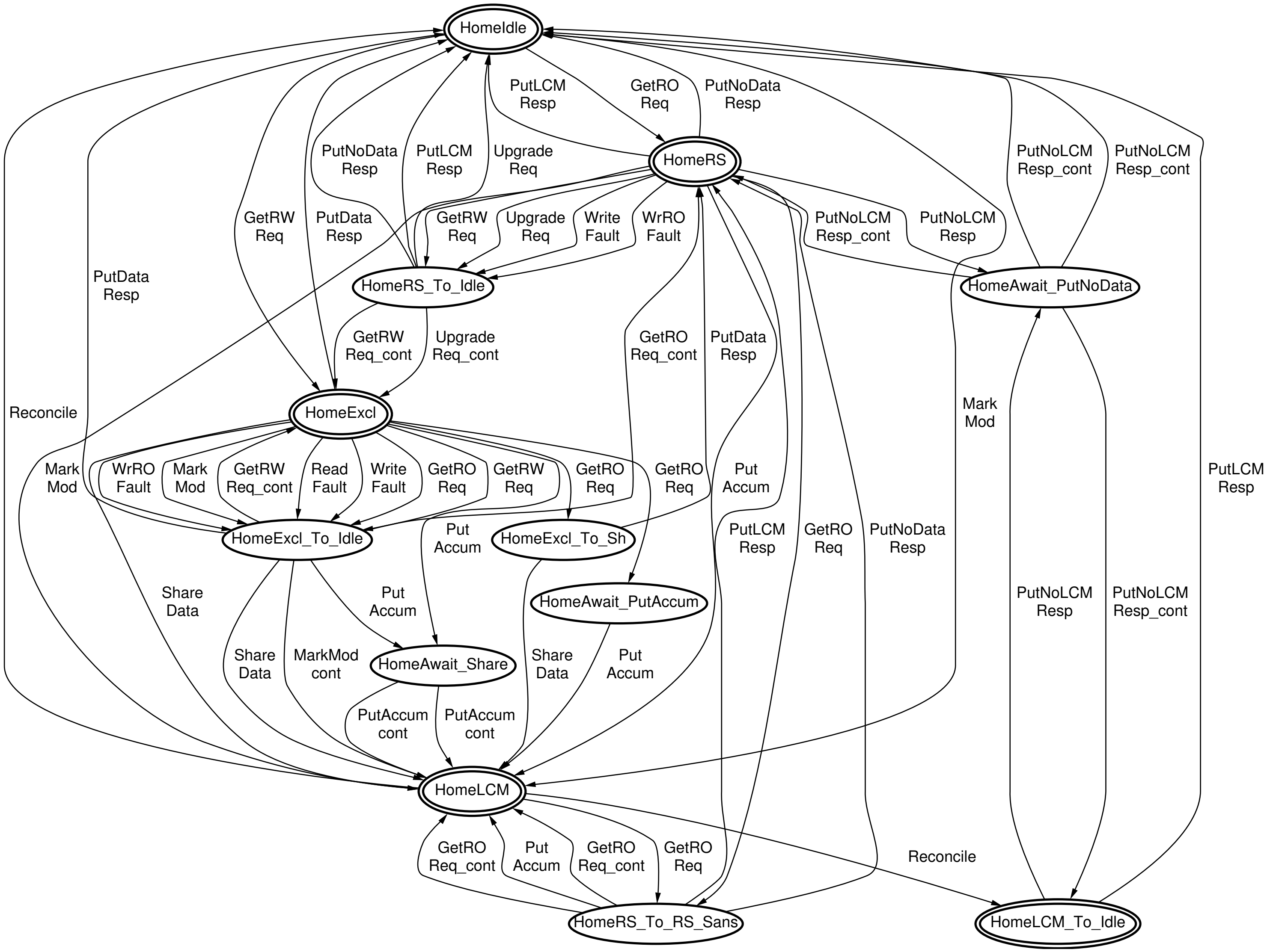


Figure A.4: LCM-MCC home-side FSM

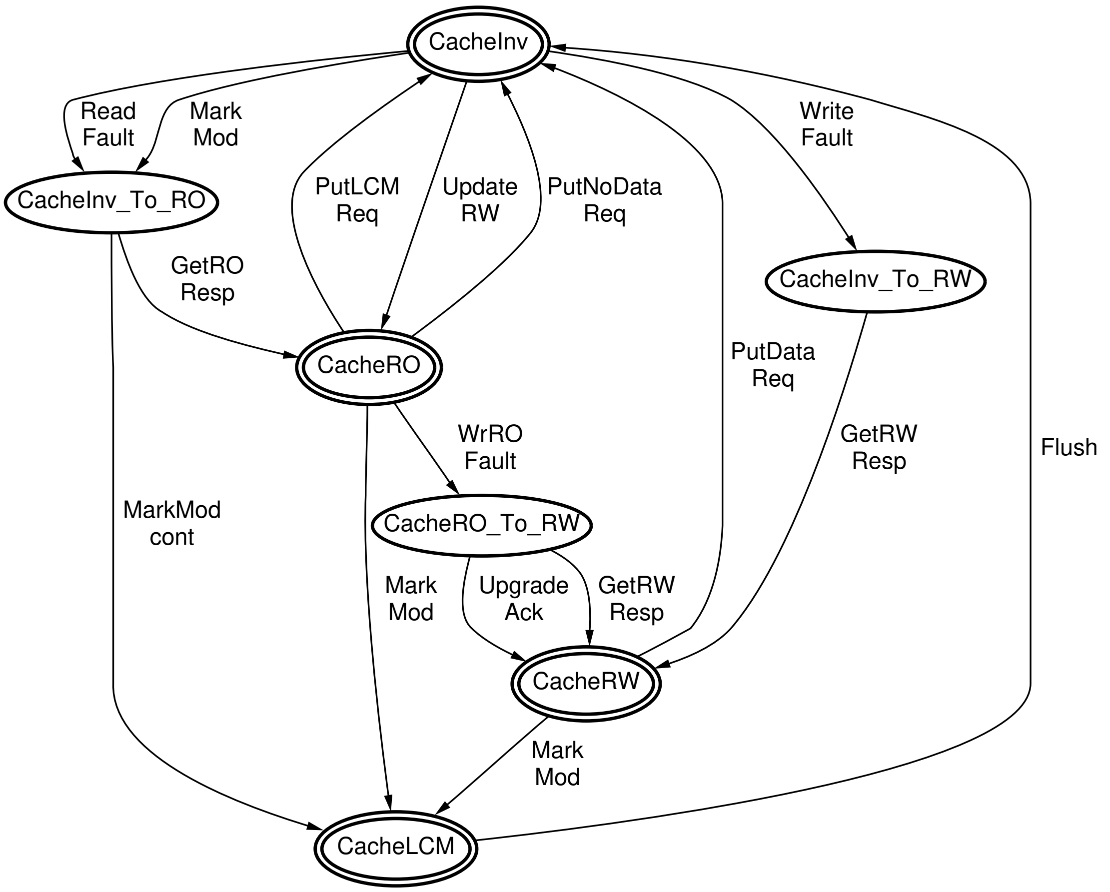


Figure A.5: LCM-SCC-Update remote-side FSM

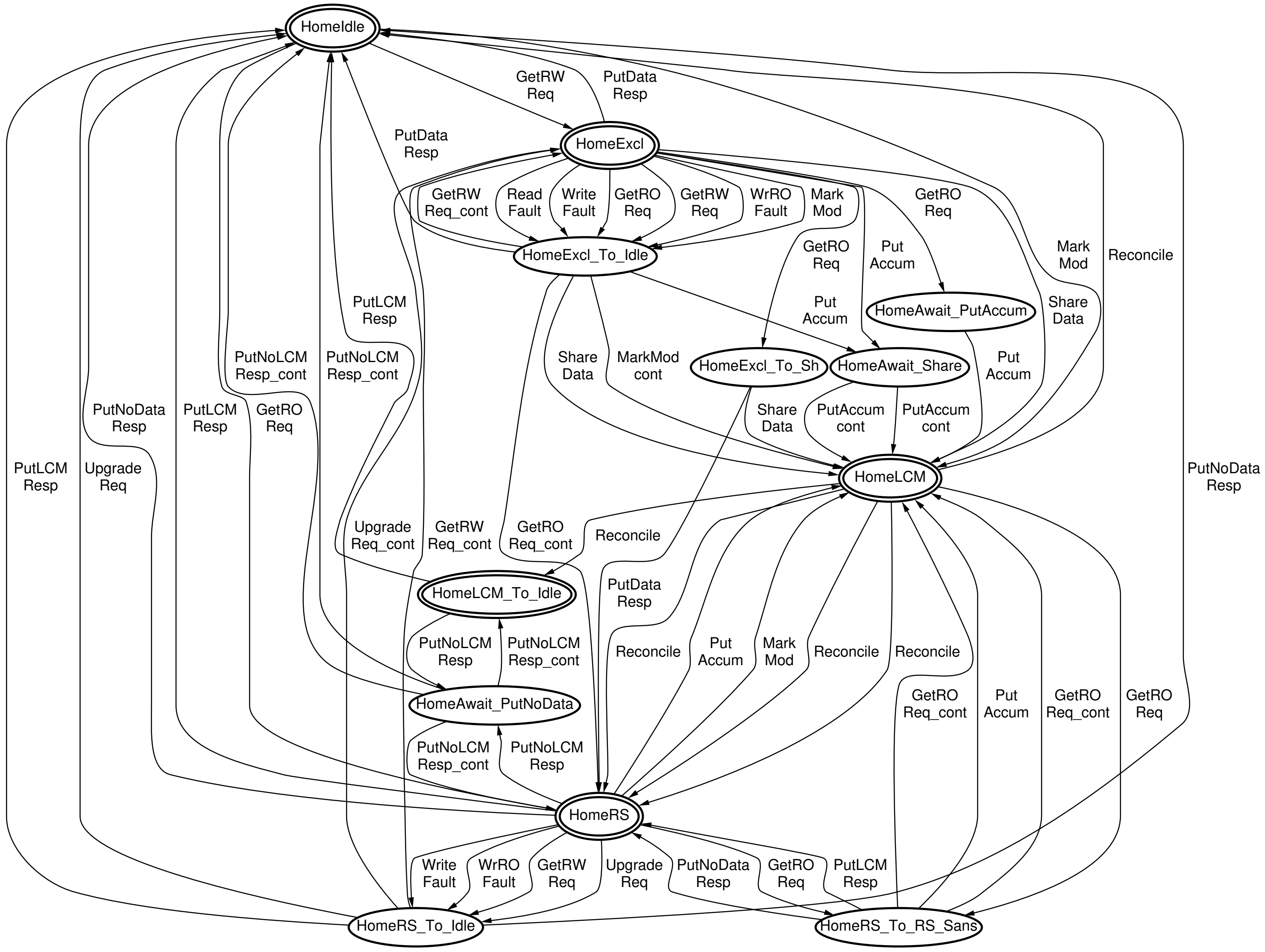


Figure A.6: LCM-SCC-Update home-side FSM

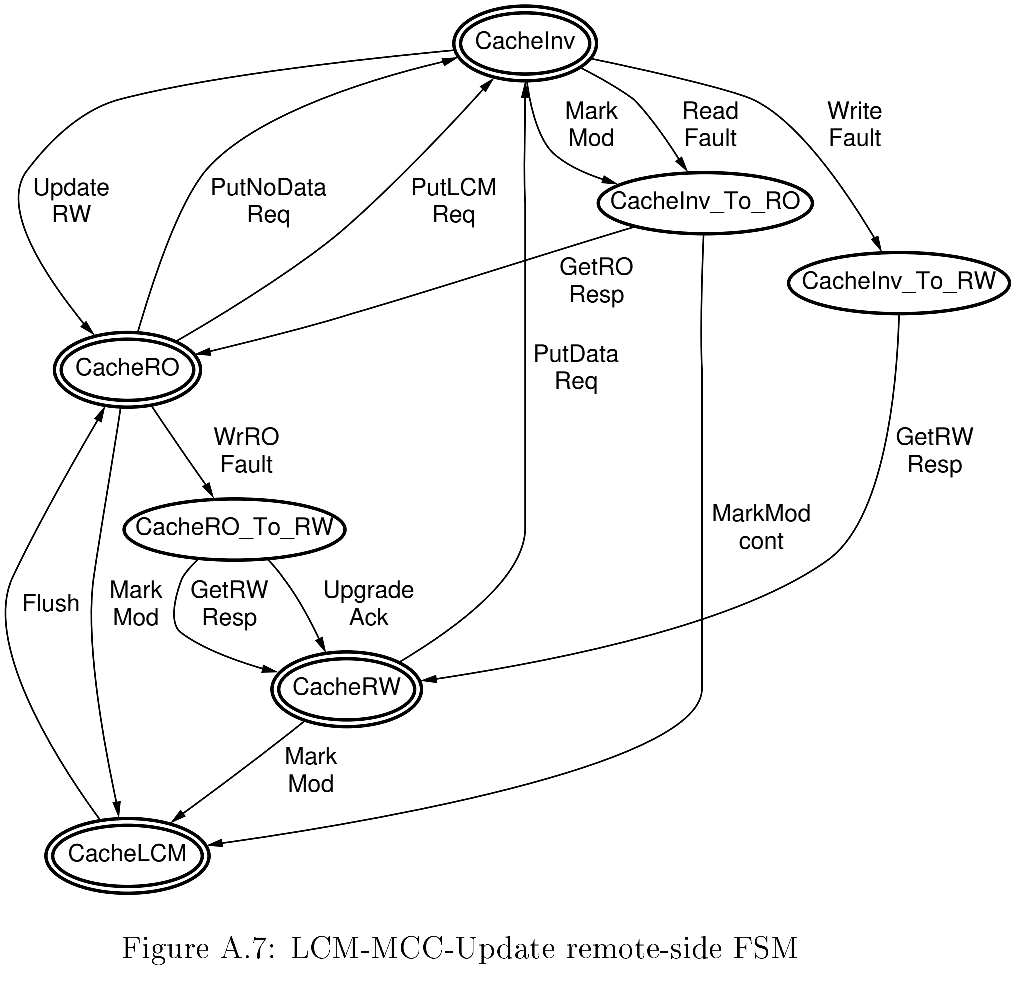


Figure A.7: LCM-MCC-Update remote-side FSM

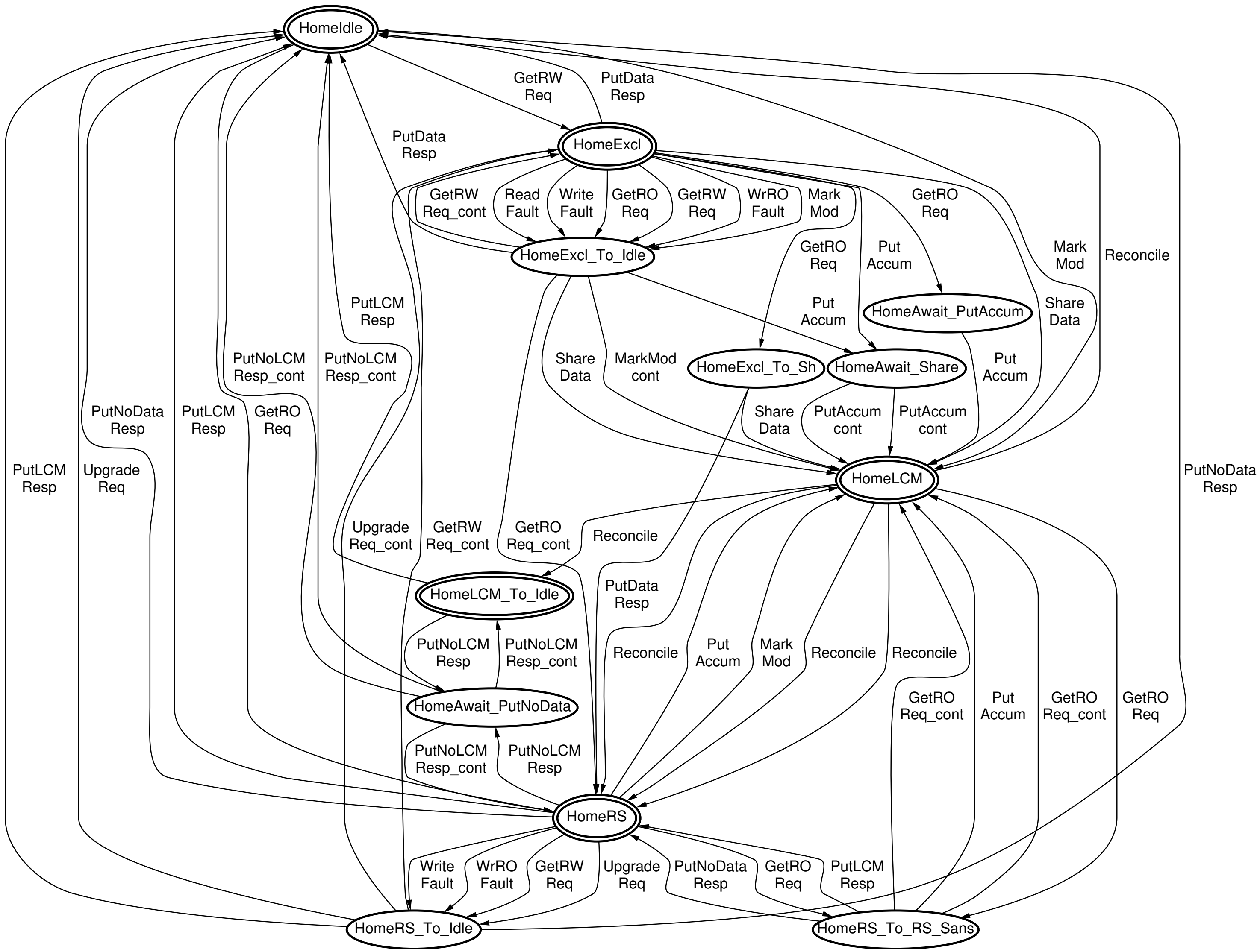


Figure A.8: LCM-MCC-Update home-side FSM

Appendix B

The Race Detection Protocols

This appendix contains the finite-state machine descriptions for the race-detection protocols. The actual handler code differs between the Race-Bit and Race-Byte protocols, but the states and transitions are identical. Thus, a single set of FSM diagrams are presented for the home and remote side. Each of the Teapot protocol specifications for the race-detection protocols are roughly 1,600 lines long.

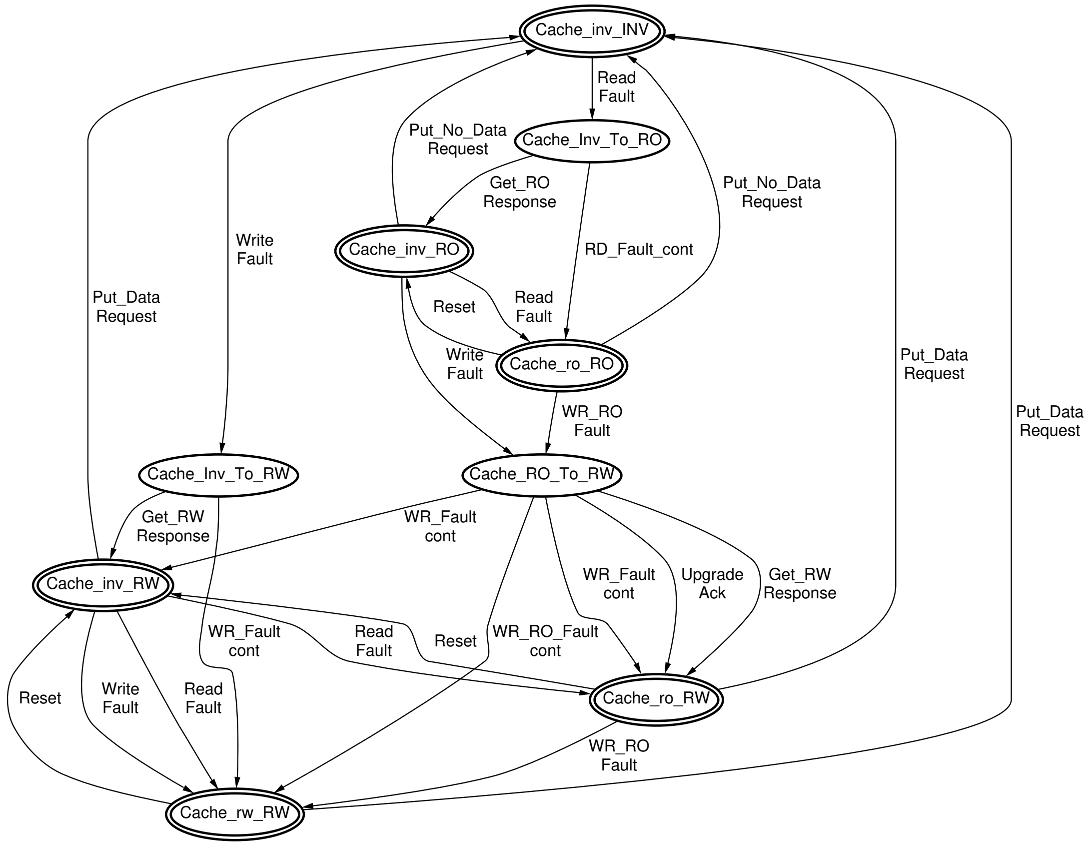


Figure B.1: Race-detection protocol remote-side FSM

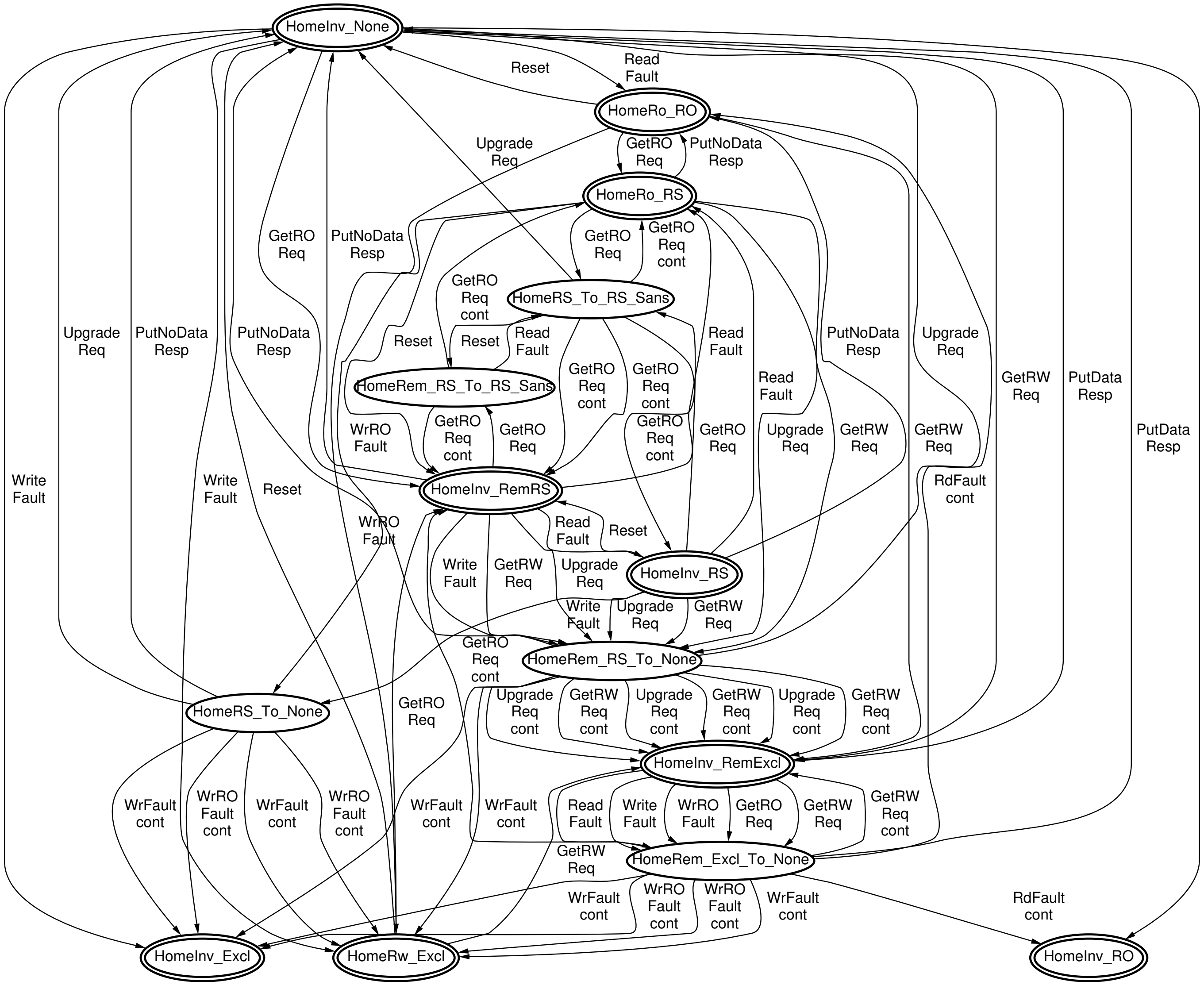


Figure B.2: Race-detection protocol home-side FSM