Appears in: "ASPLOS VI," Oct. 1994.

LCM: Memory System Support for Parallel Language Implementation*

James R. Larus Brad Richards Guhan Viswanathan

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
{larus,richards,gviswana}@cs.wisc.edu

Abstract

Higher-level parallel programming languages can be difficult to implement efficiently on parallel machines. This paper shows how a flexible, compiler-controlled memory system can help achieve good performance for language constructs that previously appeared too costly to be practical.

Our compiler-controlled memory system is called Loosely Coherent Memory (LCM). It 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 location and the way in which a processor *reconciles* multiple outstanding copies of a location. LCM memory becomes temporarily inconsistent to implement the semantics of C** parallel functions efficiently. RSM provides a compiler with control over memory-system policies, which it can use to implement a language's semantics, improve performance, or detect errors. We illustrate the first two points with LCM and our compiler for the data-parallel language C**.

*This work is supported in part by NSF PYI/NYI Awards CCR-9157366 and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, and donations from Digital Equipment Corporation, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

1 Introduction

Compiling parallel languages for parallel computers is difficult. Most of these languages assume a shared address space in which any part of a computation can reference any data. Parallel machines provide either too little or too much support for many languages [22]. On one hand, message-passing machines require a compiler to statically analyze and handle all details of data placement and access, or pay a large cost to defer these decisions to run time. On the other hand, shared-memory machines provide more dynamic mechanisms, but generally use them to implement a fixed cache-coherence policy that may not meet a language's needs.

This paper shows how a compiler can exploit control over the memory system of a parallel computer to construct a language-specific address space for a high-level parallel language. Because the address space's semantics match the language's, the compiler can generate efficient code, with assurance that the memory system will detect unusual cases and errors so a run-time system can handle them. We first present a family of memory systems, called *Reconcilable Shared Memory (RSM)*, that identifies where and how a compiler and run-time system can control memory coherence and semantics. To demonstrate this approach, we built a RSM system called *Loosely-Coherent Memory (LCM)* and used it to implement the large-grain data-parallel language C** [21].

The hardware base for RSM is a parallel computer with a Tempest-like interface, which provides mechanisms that permit user-level software to implement shared-memory policies [19, 26]. A Tempest memory system is possible on a wide range of parallel systems, including those without shared-memory hardware [30]. Tempest offers a program both control over communications and data placement, as is possible with message passing, and the dynamic fine-grain policies possible with shared memory.

Reconcilable Shared Memory (RSM) provides a global address space and basic coherence policy whose two key policies governing memory system behavior are under program control. The first is the system's response when a processor requests a copy of a cache block. The second is the system's response when a processor returns a cache block. Unlike most shared-memory systems, RSM places no restrictions on multiple outstanding writable copies of a block and permits non-sequentially consistent memory models. A language-specific coherence protocol uses RSM mechanisms to support a language's semantics directly. Custom coherence policies can also improve the performance of shared-memory programs written in any language. For example, global reductions and stale data fit naturally into the RSM model. Finally, RSM can help detect unsynchronized data accesses (data races).

We implemented a RSM system to support C**, which is a new large grain data-parallel programming language based on C++ [21]. In C**, when a program applies a *parallel function* to an aggregate data collection, the function is invoked separately on each element in the aggregate. Semantically, each function invocation executes *atomically* and *simultaneously*, so conflicting data accesses are impossible. When all invocations from an application complete, their modifications are combined into a new global state. Fortran 90 and HPF [17] whole-array operations share a similar semantics, albeit restricted to predefined functions and arrays.

Loosely-Coherent Memory (LCM) is a RSM memory system that implements C** semantics with a fine-grained copy-on-write operation and a language-specific reconciliation function. The C** compiler uses RSM directives to identify memory accesses in a parallel function that possibly conflict with accesses from other invocations. At one of these references, LCM copies the referenced cache block and makes it private to the invocation. These writable copies preserve C**'s semantics—even though memory as a whole becomes inconsistent—since other invocations do not see changes. When the parallel call terminates, LCM reconciles multiple versions of a block to a single consistent value with an application-specific reconciliation function.

We built a LCM system on Blizzard, which is a fine-grain distributed shared memory system (not a simulator) that runs at near shared-memory hardware speeds on a Thinking Machines CM-5 [30]. We compared the performance of four C** programs running under both the unmodified Stache protocol [26] and LCM (implemented using the Tempest mechanisms provided by Blizzard). We found that the LCM memory system improved performance by up to a factor of 2 for applications that used dynamic data structures. For applications with static data and sharing patterns, which a compiler could analyze and optimize, LCM's performance is comparable to a standard memory system.

The contributions of this paper are:

- A new memory model, RSM, that identifies two points in memory protocols at which compilers can control memory-system policies.
- An implementation, LCM, of this model that helps implement C**.
- A demonstration that this model and system simplify implementing a higher-level parallel language and help improve its performance.
- And, a demonstration that controlled inconsistency can help implement a parallel language.

This paper is organized as follows. Section 2 surveys related work. Section 3 explains the RSM memory model. Section 4 describes the C** language briefly. Section 5 describes LCM. Section 6 shows how the C** compiler uses LCM. Section 7 explores other applications of the memory system. Section 8 concludes the paper.

2 Related Work

Existing, scalable shared-memory machines are inflexible and provide minimal assistance for language implementation. Many use cache-coherence protocols whose policies are implemented permanently in either hardware or a combination of hardware and software [2, 11, 18, 23, 32]. These systems support only a single coherence protocol and consistency model (typically, sequential consistency) and provide few performance-enhancing mechanisms beyond prefetches and cache flushes. Compilers can circumvent coherence policies only by sending messages [19], even when language semantics or program analysis shows that much coherence traffic is unnecessary [9, 14, 22].

Relaxed consistency models trade a simple view of memory as a sequentially consistent store for increased hardware performance [1]. Most models adopt the view that memory need only be consistent at program-specified synchronization points. Relaxed consistency, instead of providing mechanisms by which a compiler or programmer can control memory semantics, takes the opposite view and leaves semantics as unspecified as possible. LCM allows memory to become inconsistent between synchronization points, but this "loose" consistency is both under program control and is an end in itself, not a byproduct of hardware optimization.

Several proposed systems, including Wisconsin Typhoon and Blizzard [26, 30] and Stanford FLASH

[20], provide low-level mechanisms to implement coherence policies in software. Typhoon and Blizzard provide the Tempest interface, which allows user-level software to implement these policies. RSM is a less general facility that explores another approach by providing a compiler with control over selective aspects of a coherence policy rather than the means to implement a new policy.

The Myrias machine implemented a copy-and-reconcile operation similar to the one in LCM, but in hardware with a fixed policy and page granularity [6].

Distributed shared memory systems, in general, provide few mechanisms for an application to control a memory system [7, 13, 24]. LCM shares with Munin [7, 10] the ability to adapt shared-memory policies to an application. Munin, however, provides a set of fixed coherence policies, each tailored for a specific sharing pattern. A programmer or compiler associates a policy with a language-level object. RSM provides a more general framework in which to develop application-specific policies by breaking coherence policies into two components, each of which is specified separately. LCM's mechanisms also apply at cache block granularity, not to entire objects or memory pages.

In VDOM [15], memory objects are immutable. Modifying an object produces a new *version* of the object. Like LCM, both systems allow multiple, distinct copies of memory to develop. VDOM handles coherence at an object level, as opposed to LCM's finer-grained cache block operations. It also uses a single coherence mechanism based on object version numbers.

The division of labor between the C** compiler and LCM system is reminiscent of the interaction between Lisp and ML systems and virtual memory [5] for stack and heap bounds checking and concurrent garbage collection [4].

Many data-parallel languages have been designed and implemented for MIMD machines, for example, C* [27], Data Parallel C [16], VCODE [12], and parts of HPF [17]. Data-parallel languages handle data races in different ways. Functional languages, such as NESL [8], avoid conflicts entirely by omitting side effects. Other languages, such as Paralation Lisp [28], ignore the problem by leaving the semantics of conflicting side effects unspecified. C* [27] uses a SIMD execution model in which all processors execute the same statement simultaneously. The array-based data parallel subset of HPF [17] specifies that all inputs to a data-parallel operation are read before any are written. C** atomic and simultaneous semantics are similar, but generalize to user-defined data-parallel functions and unstructured data.

Considerable research has been devoted to increasing grain size and removing unnecessary synchronization when compiling data parallel languages for MIMD machines [12, 16]. By contrast, C** allows a programmer to express large-grain parallelism directly, but imposes restrictions on data accesses to prohibit conflicts.

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 our LCM system fit within the RSM model. This section describes RSM and outlines why cache-coherent shared memory is an instance of this more general model.

The hardware base for RSM is a collection of autonomous processing nodes connected by a point-to-point network. Each node contains a processor, cache, network interface, and local memory. The physically distributed memory is addressed through a global address space, so each processor can refer to any location independent of where it is located. A processor manages coherence of its local memory according to the RSM policies. When a program references a memory location that is not in the processor's cache, the processor sends a request to the location's home processor for a copy of the location. The home processor responds with this location, which may be marked read-only or read-writable. The transfer unit is called a *block*. A processor eventually returns a (possibly modified) location to its home processor for a variety of reasons, such as a cache replacement or a program-initiated cache flush, or in response to a request from another processor.

As described above, RSM assumes the same basic mechanisms as cache-coherent shared memory [26, 32] but generalizes the policies. RSM is a family of protocols that 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 [11, 23], 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.

4 C** Language

We implemented the memory model described previously and used it to support programs written in C**, a new data-parallel language [21]. This section provides a brief overview of the language and a sample C** program.

4.1 Overview of C**

C** is a data-parallel programming language that extends C++ with a small number of features. It supports a new form of data parallelism (large-grain data parallelism), that offers much of the semantic simplicity of SIMD data parallelism [27], but does not require lockstep execution. In the C** data-parallel model, parallelism results from applying a *parallel function* across a collection of data called an *aggregate*. Aggregates look and behave like C++ arrays, but form the basis for parallel functions. Applying a parallel function creates an asynchronously executed *parallel function invocation* for each element in an aggregate.

Parallel functions are defined like other C++ member functions. The mandatory aggregate argument to a parallel function is specified by the keyword `parallel`. Additional features of parallel functions include:

- Pseudo variables (#0, #1, etc.) that specify on which element in an aggregate a particular parallel invocations is operating.
- The ability to combine values from parallel function invocations or from a reduction on a location.

4.2 Parallel Functions Semantics

Parallel invocations created by applying a parallel function to an aggregate execute asynchronously and independently of one another. In an imperative language, such as C**, with a global address space, asynchronous execution raises the possibility of data access conflicts and races. In C**, parallel function invocations must appear to execute "atomically and simultaneously." In other words, when an invocation modifies a global location, the modification is private to the invocation and cannot be seen by other, concurrently executing invocations. When all parallel function invocations complete, the program's global state is updated by merging all private modifications. In general, if two or more invocations modify the same location, exactly one modified value will be visible after this merge. C** allows nested parallel functions (i.e., parallel calls from within parallel functions), but this paper considers only non-nested parallel functions.

To make these ideas concrete, below is a trivial C** function that illustrates C** semantics. The function computes a stencil on interior points of a two-dimensional matrix:

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

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

The function invoked at each array element reads the values of its four neighbors and writes its own value. If memory reads and writes follow a simple shared-memory model, a programmer would not know if a neighboring value had been updated. In C**, conflicts like this are impossible, because all shared modifications, including the one above, do not become visible outside of their local invocation until the parallel function invocation completes. Since `stencil` writes each data point only once, no modifications conflict and the merge phase only records the value assigned to a location.

C** also has *reduction assignments*, which combine the values written into a location with a binary, associative operator and leave the final value in the modified location. For example, to sum the elements in array `A` into a variable `total`, we would write:

```
double total;

void sum(parallel Matrix &A) parallel
{
    total %+= A[#0][#1];
}
```

In contrast, the simple assignment statement `total += A[#0][#1]` adds each element of `A` to a *local* copy of `total`. The merge phase combines these local values into a single value by applying the appropriate reconciliation function.

5 LCM

RSM systems can aid in implementing parallel programming languages, particularly for higher-level languages such as C**. A natural way to implement C**'s semantics is a copy-on-write scheme, in which each parallel invocation obtains and modifies its own copy of shared data. We implemented this policy in a RSM system called Loosely Coherent Memory (LCM). LCM and the C** compiler cooperate to detect the need for shared data and to copy it, instead of the conventional approach in which a compiler generates conservative code to copy shared data. LCM's copies, although they share the address of the original, are private to a processor and remain inconsistent until a global reconciliation returns memory to a consistent state.

RSM offers several advantages over explicit copying. A compiler can produce code optimized for when no copying is necessary, which is likely to predominate in most programs.[1] Compiler-produced copying code is conservative and incurs unnecessary overhead either by copying too much data or by testing to avoid unnecessary copying. The LCM copy-on-write scheme defers copying until a location is accessed, which reduces the quantity of data that must be copied.

A compiler's control of LCM permits optimizations when analysis is possible. For example, not all modifications to shared data need cause a copy. Only items possibly shared between processes must be copied. If compiler analysis determines that no other process will access a location, it need not be copied, which avoids the overhead of making and reconciling a copy. However, this approach requires close cooperation between the compiler and memory system to select—at a fine-grain—policies governing portions of data structures.

[1] Languages can encourage programmers to avoid data races. C**, for example, offers no guarantees about the value resulting from multiple modifications of a location or even if parts of a large object will be written by the same invocation.

In C**, computation alternates between parallel and sequential phases. Memory becomes coherent at the end of a parallel phase as processors reconcile their modified memory locations. C** semantics dictate how copies are reconciled. In general, C** requires only that the coherent value left in a memory location modified by a parallel function call be a value produced by one invocation of the call. LCM discards all but one of the modified copies. However, values written by C**'s reduction assignments (Section 4.2) require different reconciliation functions that combine values.

5.1 LCM Implementation

We implemented a LCM system on Blizzard, which is a fine-grain distributed shared memory system that runs at near shared-memory hardware speeds on a Thinking Machines CM-5 [30]. Blizzard implements the Tempest interface [26], which permits user-level control of the memory system. LCM is user-level code that runs on the CM-5. We started with the user-level Stache protocol [26], which provides cache-coherent shared memory and uses a processor's local memory as a large, fully associative cache. This cache is essential to ensure that a processor's locally modified (inconsistent) blocks are not lost by being flushed to their home node. When a modified cache block is selected for replacement (either because of a capacity or conflict miss), the block is moved to the Stache in local memory. On a cache miss to the block, its value comes from the Stache, rather than its home processor.

LCM provides the C** compiler with three directives. The first, `mark_modification(addr)`, creates an inconsistent, writable copy of the cache block containing `addr`. If the block is not already in the processor's cache, it is brought in. 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 coherent state. This directive flushes all modified blocks back to their home processors to be reconciled. Outstanding read-only copies of these blocks are then invalidated throughout the system. The third, `flush_copies()`, performs a partial reconciliation by flushing a processor's modified cache blocks back to their home processors. The next section illustrates how the C** compiler uses these directives.

Reconciliation occurs at the home location of a modified block, when it returns. This poses a potential bottleneck for systems with many processors, but it is unlikely to be a problem for several reasons. First, a typical block is modified by relatively few processors and so few copies need to be reconciled.

Second, copies are flushed as each invocation completes, so few blocks arrive simultaneously. Finally, if reconciliation became a bottleneck on large systems, the process could be organized as a tree-structured reduction.

C** parallel function invocations start execution with the original (pre-parallel call) global state. LCM retains an unmodified copy of global data throughout a parallel call. At the first write to a cache block managed by the copy-on-write policy, the block's home node creates a *clean copy* of the block in main memory. The node uses a clean copy to satisfy subsequent requests for unmodified global data.

Another complication is that each processor typically runs many distinct invocations of a parallel function. The system must ensure that a new invocation does not access local cache blocks modified by a previous invocation. To avoid this error, LCM's `flush_copies()` directive removes modified copies of global data from the Stache. If a compiler cannot ensure that invocations access distinct locations, it issues this directive between invocations. This directive flushes cache blocks to their home processor, where they are reconciled. A subsequent read of one of these blocks returns its original value from the clean copy. Cache flushing, although semantically correct, performs poorly for applications that reuse data in flushed blocks. In another approach, each processor keeps a clean copy of every block it modifies. In this case, the `flush_copies` directive returns modified values to their home node and replaces the cached value with the clean copy, so it remains local for a subsequent reference.

LCM's memory usage depends on the number of potentially modified locations. At a location's first `mark_modification` directive, LCM creates a clean copy in memory. Cached copies resulting from this directive require slightly more state information than ordinary cached blocks. Clean copies exist only during a parallel function call and are reclaimed at the `reconcile_copies()` directive.

6 Compiling C**

Compiling a C** program to run under LCM is straightforward. To ensure the correct semantics for parallel functions, the C** compiler inserts memory system directives, described above, in parallel functions. Alternatively, the compiler could guarantee these semantics with run-time code that explicitly copies data potentially modified in a parallel function invocation. Explicit copying works well for functions with static and analyzable data access patterns. However, it becomes complicated and expensive for programs with dynamic behavior since the generated code must either perform run-time checks or copy a conservative superset of the modified locations. This section illustrates both approaches with a static parallel function (the stencil function) and a dynamic parallel function (an adaptive mesh) and compares the performance of LCM against the explicit copying strategy.

6.1 Stencil

As an example, consider a simplified version of the code generated by the C** compiler for a `stencil` function (Section 4.2) to run under LCM:

```
void stencil_SPMD(parallel matrix &A)
{
  for all invocations assigned to me do
  {
    Set variables #0 and #1;

    // Function body:
    int x = #0, y = #1;

    mark_modification(A[x][y]); // LCM directive

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

Each invocation writes to `A[x][y]`, which is also read by its four neighboring invocations. Compiler analysis easily detects this potential conflict, which the C** compiler rectifies with `mark_modification` directives. 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.

Because compiler analysis reveals that `stencil` accesses the entire array, the C** compiler could also preserve C** semantics by maintaining two copies of `A`—all reads come from the old copy of `A` and all writes go to the new copy of `A`. After each iteration, the code exchanges the two arrays with a pointer swap. This simple technique preserves the C** semantics with little overhead beyond the cost of twice the memory and cache usage.

6.2 Dynamic C** Program

LCM offers greater benefits for programs with dynamic behavior that is difficult or impossible to analyze. These programs require extensive (and expensive) run-time operations to run in parallel [29]. For example, consider an adaptive mesh version of `stencil`, which selectively subdivides some mesh points into finer detail. It is part of a program that

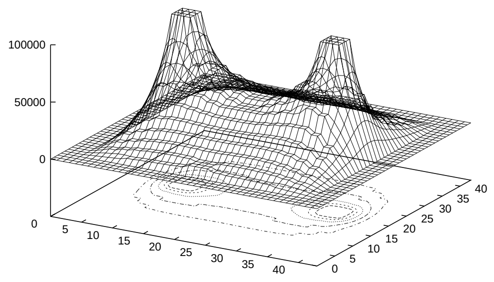

Figure 1: Relaxation of an adaptive mesh

computes electric potentials in a box. The program imposes a mesh over the box and computes the potential at each point by averaging its four neighbors. At points where the gradient is steep, finer detail is necessary and the program subdivides the cell into four child cells. This process iterates until the mesh relaxes (see Figure 1). Initially, points on the mesh are represented in a two-dimensional matrix, but dynamically-allocated quad-trees capture cell subdivision:

```
// Update my quad-tree in the mesh
//
double Mesh::update_mesh() parallel
{
  // What part of tree changed?
  *self = update_quad_tree(self, neighbors);

  // Return maximum of local values
  return %> local_epsilon;
}

// Main program - do the iterations
//
main()
{
  ...
  create_mesh();
  while (difference >= epsilon)
      difference = update_mesh();
  ...
}
```

In this program, the mesh changes dynamically so a compiler cannot determine which parts will be modified. Without a LCM system, a compiler must conservatively copy the entire mesh between iterations to ensure C**'s semantics. With LCM, the memory system detects modifications and copies only data that is modified.

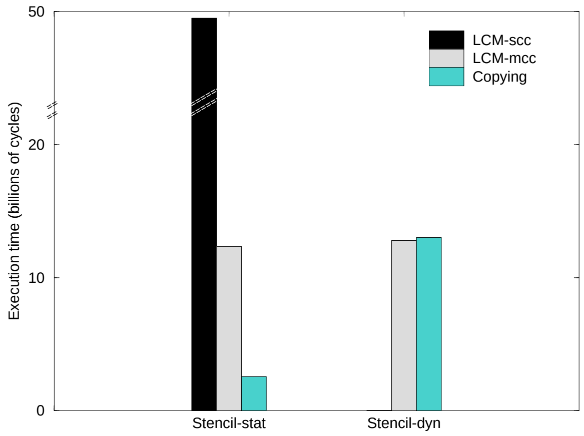

Figure 2: Stencil execution time.

Program	Cache misses (in thousands)			Clean copies (in thousands)	
	scc	mcc	Copying	scc	mcc
Stencil-stat	3,216	6,374	1,035	13	406
Stencil-dyn		6,615	12,696		6,541
Adaptive	4,427	3,335	2,245	66	2,398
Threshold	411	116	432	2	63
Unstructured	1,168	1,156	1,176	0	130

Table 1: Benchmark cache misses and clean copies.

6.3 Performance

We evaluated the performance of LCM with four small C** benchmarks (*Stencil*, *Adaptive*, *Threshold*, and *Unstructured*) that performed similar computations with varying degrees of dynamic behavior. As a baseline, we ran the same C** programs, compiled under an option to perform explicit copying, on the unmodified Stache protocol [26]. Both LCM and Stache run under Blizzard-E [30] on a CM-5 with 32 processors.

We measured two versions of LCM. The first, *LCM-scc*, keeps a single clean copy of each marked cache block at the block's *home* node. Section 5.1 explains why this approach can perform poorly for applications with spatial or temporal reuse between invocations. The other system, *LCM-mcc*, maintains a clean copy of a marked block on all processors that obtain the block. When a modified cached copy is flushed home, after a parallel function invocation, the block is reinitialized immediately from the local clean copy.

Figures 2 and 3 show the execution time for the four test programs and three memory systems. Table 1 records cache misses and clean copies. Below, we discuss the test programs in more detail.

Stencil performs a simple, regular four-point stencil computation over a fixed mesh (Section 6.1). The data is for 50 iterations on a 1024x1024 mesh. Since

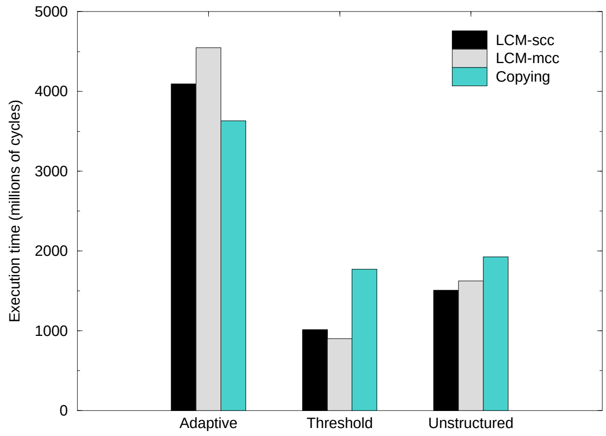

Figure 3: Benchmark execution time.

a cache block holds eight single-precision floats, the program has considerable spatial cache-block reuse. LCM-scc does not exploit this reuse and its performance is roughly four times slower than LCM-mcc, which reduced cache misses by a factor of almost 8 over LCM-scc.

We measured two versions of this program. The first (*Stencil-dyn*) dynamically partitioned the mesh into P (number of processor) chunks at the beginning of an iteration. The second (*Stencil-stat*) only partitions the mesh once, at the start of the computation. The first version ran roughly 2% faster under LCM-mcc than under Stache. The second version ran almost five times faster under Stache because the static partition enabled this protocol to keep the entire interior of a chunk in a processor's local memory throughout the computation and only communicate boundary elements. On a machine with a limited cache or on problems that require less repeatable scheduling techniques, the first version's performance is likely to be more typical.

Adaptive is another stencil computation, but over a time-varying mesh (Section 6.2). With a conventional memory system, like Stache, the program maintains two copies of the mesh and copies values between them before an iteration. By contrast, LCM's fine-grain copy-on-write policy copies only modified locations. The measurements were collected for 100 iterations on an initial 64x64 mesh, with a maximum quad-tree depth of 4. Statically and dynamically partitioned version were tested. For the static code, LCM-scc ran slightly (1.1%) faster than LCM-mcc because spatial reuse in the quad trees is too limited to justify the more costly protocol. Both LCM versions ran 13% slower than the statically-scheduled Stache version for reasons discussed above. With dynamic scheduling, the benchmark running on LCM-mcc is almost two times faster (92%) than the Stache version.

Threshold performs a stencil computation over a structured 512x512 mesh (50 iterations), but does not modify all mesh elements. At each point, *Threshold* reads its neighbors and updates the point only if its value changed by more than a threshold. Without LCM support, the mesh must be completely copied in each iteration to move values from the old mesh to the new one. The program itself copies values that are not updated, so a separate copying phase is unnecessary. LCM only copies modified values. The ratio of modified to unmodified mesh cells is small (2.1%) since the mesh is initially zero except for a few points. Only cells near a fixed value change during the first iterations. LCM-mcc is 12% faster than LCM-scc because of spatial reuse of single-precision floats. LCM ran considerably faster than Stache (97% and 74%, respectively) because it copied far fewer locations.

Unstructured applies a relaxation technique to an unstructured mesh. This program builds the graph (256 nodes and 1024 edges, 512 iterations) and statically partitions it. To ensure C** semantics without LCM support, the program maintains an extra copy of the nodes. No additional copying is necessary since all nodes are updated in each iteration. *Unstructured* differs from *Stencil* because *Unstructured* has little locality due to its irregular structure. LCM-mcc's performance exceeds LCM-scc's by 8% because of spatial reuse. LCM is faster, by 19% to 28%, than Stache because the graph data structure has many cross-processor edges that cause communication under this protocol as well as LCM.

These measurements show that LCM helps efficiently implement C**, particularly for programs whose behavior is difficult to analyze or predict statically. Few compiler techniques have addressed these programs, which are typically written by hand, with the aid of elaborate libraries [29]. The numbers above also show that a compiler should not rely exclusively on LCM. In situations, such as *Stencil-stat*, in which communication is simple and rarely occurs under a protocol like Stache, LCM has little to offer. One of the virtues of user-level shared memory [26] is that a compiler can make this choice (or even, use both in a program) by selecting the libraries linked with a program.

7 Other LCM Applications

Reconcilable shared memory proved useful in implementing C** parallel functions. But RSM is not limited to this role. RSM can support other language features, both for C**-style languages and languages with more traditional semantics. This section outlines a few other applications for RSM.

7.1 Reductions

The reconciliation mechanism in RSM combines data values from multiple copies of a cache block to produce a single value for the block. This mechanism can also implement reductions in conventional languages. For example, a programmer typically sums the elements in an array by adding them into a variable:

```
for i := 1 to N do
  total = total + a[i];
```

To run this loop in parallel, the programmer could protect `total` with a lock, which would introduce a bottleneck, or could rewrite the loop so each processor reduces its portion of the array into a private variable and then sum these partial results. In a RSM system, a compiler that detects the reduction in the loop [3] could choose a reconciliation function for `total`'s cache block that sums the values added to the location's initial value. When the loop completed, the locally cached accumulators are reconciled into a single value.

Of course, if the compiler can detect the recursion and emit RSM directives, it can call a library routine or produce code similar to that outlined above. The advantages of RSM are two-fold. First, it requires no additional compiler analysis to distinguish multiple accumulators in complex situations in which program analysis fails, for example, in reductions over a pointer-based data structure such as an unstructured mesh or code like: $\mathrm{A}[f(i)] = \mathrm{A}[f(i)] + \mathrm{c}$. Second, RSM is likely to be less expensive than shared-memory code since RSM can implement the reduction by sending messages rather than communicating through memory locations [14, 19].

7.2 Semantic Violation Detection

A RSM system can help implement other parallel languages. For example, Steele proposed a language semantics that forbids programs with conflicting or racing side effects [31]. Static analysis is too conservative since it may disallow programs that do not violate the semantics. Steele proposed a scheme for detecting run-time violations that required maintaining an access history for each memory location that could possibly communicate side-effects from one processor to another. Whenever an operation accesses a region, it is added to the region's history and checked against the previous accesses to the region to detect conflicts. Optimizations can decrease the space for access histories, but the worst-case remains unbounded.

LCM can identify illegal programs without access histories. A program uses the LCM mechanisms to make copies of modified shared data, just as for C**. At synchronization points, the program forces a reconciliation that detects conflicts (since multiple modifications are prohibited, values need not be reconciled). If a word in a block is modified by multiple processors, a conflict occurred. A processor can record the modified words in a cache block by setting its access control to *ReadOnly* [30] and trapping stores until all words are modified.[2]

A read-write semantic violation occurs if a memory location is concurrently read by one processor and written by another. When reconciliation is performed, modified cache blocks are returned to their home node to be combined. The home node is also aware of read-only copies. A check during reconciliation can ensure that readable and writable copies of the block were not outstanding at the same time. However, outstanding read-only copies need not be *used* during the parallel phase of the computation. A read-only block could remain in a processor's cache from a previous computation. To catch actual violations, all read-only cache blocks must be flushed from the caches at synchronization points.

7.3 Data-Race Detection

The scheme described above also detects unsynchronized data accesses. A memory system could perform this service for languages with more traditional semantics, thereby detecting data races at run time. The same tradeoffs discussed above apply to race detection. Potential data access conflicts could be identified by performing reconciliations at each synchronization point and detecting the coexistence of read-only and writable copies of a block. Detecting actual races would require flushing all read-only cache blocks from the caches after each reconciliation. The loss in performance is less critical here since a run-time race detection system is most likely used only while debugging, and other run-time techniques have high overheads [25].

7.4 Reducing False Sharing

False sharing occurs when several processors concurrently access (with at least one write) different locations in the same cache block. LCM-like systems can reduce the effects of false sharing. If multiple processors modify distinct locations in a block, each process can have its own copy of the block and compute without contending for access. In general, this scheme must be limited to memory accesses that a compiler or programmer ensures are non-conflicting.

[2] Many thanks to Anne Rogers for this suggestion.

7.5 Stale Data

In some scientific applications, such as N-body simulations, contributions from distant elements are less significant than those of closer elements. Repeatedly using old information about distant elements may not adversely affect the computation. This optimization is particularly attractive for large systems, in which the number of outstanding values and the cost of updating them is large.

In a cache-coherent machine, keeping around "stale" data requires explicit copying. When a value's producer changes a location, its read-only copies are updated or invalidated, even if consumers would be content with the old value. A programmer can copy data to private memory, but this complicates the program and increases the required storage. An RSM system would allow a program to operate with stale data for many iterations before obtaining the latest value. The data's consumer uses an operation similar to `mark_modification` to create a local copy of a block. The consumer obtains the most recent value by reconciling its copy with the producer's value. The consumer may not know the producer's identity, in which case the consumer can simply flush the block. The next reference will bring its latest value back into the cache.

8 Conclusion

In this paper, we showed that a flexible, program-controlled memory system can help a compiler achieve good performance for a language feature that previously appeared too complex and costly to be practical. We explored this new approach to parallel language implementation in the context of the data-parallel language C** and a new memory system called Loosely Coherent Memory, which cooperates closely with our C** compiler. The compiler produces code that assumes that data access conflicts do not arise. If these conflicts occur, the LCM system detects them and invokes run-time code to handle them by copying the accessed block. This division of labor simplifies the compiler, which can concentrate on the expected case rather than worry about the worst case.

More generally, we presented a new model called Reconcilable Shared Memory, of which LCM is an example. RSM provides a program with control over two aspects of a coherence protocol: producing copies of a block and reconciling the copies. The RSM model is not limited to compiler support for higher-level languages. In this paper, we briefly described how RSM can be used to improve the performance of general shared-memory programs with efficient global reductions, reduced false sharing, and stale data. It also has potential applications in run-time race detection.

Acknowledgements

This work was performed as part of the Wisconsin Wind Tunnel project, which is co-lead by Profs. Mark Hill, James Larus, and David Wood and funded by the National Science Foundation. We would like to thank Sarita Adve, Anne Rogers, and Guy Steele for helpful comments on this research and earlier drafts of this paper.

References

[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, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[3] Zahira Ammarguellat and W.L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*, pages 283–295, June 1990.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 11–20, June 1988.

[5] 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.

[6] 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.

[7] John K. Bennett, John B. Carter, and Willy Zwanepoel. 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.

[8] Guy E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical Report CMU-CS-93-129, Department of Computer Science, Carnegie Mellon University, April 1993.

[9] William J. Bolosky and Michael L. Scott. False Sharing and its Effect on Shared Memory Performance. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, September 1993.

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

[11] 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.

[12] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fischer. Size and Access Inference for Data-Parallel Programs. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 130–144, June 1991.

[13] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[14] 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*, November 1994. To appear.

[15] Michael J. Feeley and Henry M. Levy. Distributed Shared Memory with Versioned Objects. In *OOPSLA '93: Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 247–262, October 1992.

[16] Phillip J. Hatcher, Michael J. Quinn, Anthony J. Lapadula, Bradley K. Seevers, Ray J. Anderson, and Robert R. Jones. Data-Parallel Programming on MIMD Computers. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):377–383, July 1991.

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

[18] 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 ASPLOS V, Oct. 1992.

[19] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[20] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[21] 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, New Haven, August 1992. Springer-Verlag.

[22] James R. Larus. Compiling for Shared-Memory and Message-Passing Computers. *ACM Letters on Programming Languages and Systems*, 2(1–4):165–180, March–December 1994.

[23] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, June 1990.

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

[25] Robert H.B. Netzer and Barton P. Miller. Improving the Accuracy of Data Race Detection. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 133–144, April 1991.

[26] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[27] John R. Rose and Guy L. Steele Jr. C*: An Extended C Language for Data Parallel Programming. In *Proceedings of the Second International Conference on Supercomputing*, pages 2–16, Santa Clara, California, May 1987.

[28] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, 1988.

[29] 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.

[30] 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)*, October 1994. To appear.

[31] 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.

[32] 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 *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.