CRL: High-Performance All-Software Distributed Shared Memory[*]

Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper introduces the C Region Library (CRL), a new all-software distributed shared memory (DSM) system. CRL requires no special compiler, hardware, or operating system support beyond the ability to send and receive messages. It provides a simple, portable shared address space programming model that is capable of delivering good performance on a wide range of multiprocessor and distributed system architectures.

We have developed CRL implementations for two platforms: the CM-5, a commercial multicomputer, and the MIT Alewife machine, an experimental multiprocessor offering efficient support for both message passing and shared memory. We present results for up to 128 processors on the CM-5 and up to 32 processors on Alewife. In a set of controlled experiments, we demonstrate that CRL is the first all-software DSM system capable of delivering performance competitive with hardware DSMs. CRL achieves speedups within 30% of those provided by Alewife's native support for shared memory, even for challenging applications (Barnes-Hut) and small problem sizes.

1 Introduction

This paper introduces the C Region Library (CRL), a new all-software DSM system for message-passing multicomputers and distributed systems. The challenge in building such as system lies in providing the ease of programming afforded by shared memory models without sacrificing performance or portability. Parallel applications built on top of CRL share data through *regions*. Each region is an arbitrarily sized, contiguous area of memory defined by the programmer. Regions are cached in the local memories of processors; cached copies are kept consistent using a directory-based coherence protocol. Because coherence is provided at the granularity of regions, instead of memory pages, cache lines, or some other arbitrarily chosen fixed-size unit, CRL avoids the concomitant problems of false sharing for coherence units that are too large or inefficient use of bandwidth for coherence units that are too small. We show that CRL is capable of delivering performance competitive with hardware-supported DSM systems.

Three key features distinguish CRL from other software DSM systems. First, CRL is system- and language-independent. Providing CRL functionality in programming languages other than C should require little work. Second, CRL is portable. By employing a region-based approach, CRL is implemented entirely as a library and requires no functionality from the underlying hardware or operating system beyond that necessary to send and receive messages. Third, CRL is efficient. Very little

[*] Technical Report LCS-TM-517, MIT Laboratory for Computer Science, March 1995.

software overhead is interposed between applications and the underlying message-passing mechanisms. While these features have occurred in isolation or in tandem in other software DSM systems, CRL is the first software DSM system to provide all three in a simple, coherent package.

Because *shared address space* or *shared memory* programming environments like CRL provide a uniform model for accessing all shared data, whether local or remote, they are relatively easy to use. In contrast, message-passing environments burden programmers with the task of orchestrating all interprocessor communication and synchronization through explicit message passing. While such coordination can be managed without adversely affecting performance for relatively simple applications (*e.g.,* those that communicate infrequently or have relatively simple communication patterns), the task can be far more difficult for large, complex applications, particularly those in which data is shared at a fine granularity or according to irregular, dynamic communication patterns [38, 39].

In spite of this fact, message passing environments such as PVM [16] and MPI [29] are often the *de facto* standards for programming multicomputers and networks of workstations. We believe that this is primarily due to the fact that these systems require no special hardware, compiler, or operating system support, thus enabling them to run entirely at user level on unmodified, "stock" systems. Because CRL also requires minimal support from the underlying system, it should be equally portable and easy to run on different platforms. As such, we believe that CRL could serve as an excellent vehicle for applications requiring more expressive programming environments than PVM or MPI.

We have developed an implementation of CRL for Thinking Machines' CM-5 family of multiprocessors. Because today's networks of workstations offer interprocessor communication performance rivaling that of the CM-5 [6, 30, 41, 43], we believe that the performance of our CRL implementation for the CM-5 is indicative of what should be possible for an implementation targeting networks of workstations using current technology. Using the CM-5 implementation of CRL, we have run applications on systems with up to 128 processors.

CRL is the first all-software DSM system capable of delivering performance competitive with hardware DSMs. To demonstrate this fact, we ported our CRL implementation to the MIT Alewife machine [1]. Since Alewife provides efficient hardware support for both message passing and shared memory communication styles [23], the performance of applications running under CRL (using only message passing for communication) can be readily compared to the performance of the same applications when hardware-supported shared memory is used instead. Such experiments are *controlled*—only the communication interface used by the programming system is changed: the processor, cache, memory, and interconnect architectures remain unchanged. CRL achieves speedups within 30% of those provided by Alewife's native support for shared memory, even for challenging applications (Barnes-Hut) and small problem sizes.

The rest of this paper is organized as follows: Section 2 describes a framework for classifying DSM systems in terms of how three basic mechanisms required to provide a shared memory abstraction are implemented. Section 3 describes the goals, programming model, and implementation of CRL. Section 4 provides details about the experimental platforms used in this research (CM-5, Alewife). Section 5 presents performance results for CRL and compares them with Alewife's native shared memory support, both in terms of low-level features and delivered application performance. Section 6 recaps the major points of the paper, discusses their implications, and identifies some areas for future work. Section 7 provides a brief overview of related work. Finally, in Section 8, we draw our conclusions.

2 Mechanisms for DSM

This section presents a framework for classifying and comparing DSM systems. This classification scheme is primarily intended for use with DSM systems that employ a *data shipping* model in which threads of computation are relatively immobile and data items (or copies of data items) are brought to the threads that reference them. Other types of DSM systems (*e.g.*, those in which threads of computation are migrated to the data they reference) are also possible [4, 7, 13, 19]; a suitably generalized version of the classification scheme presented here could likely be applied to these systems as well.

We classify systems by three basic mechanisms required to implement DSM and whether or not those mechanisms are implemented in hardware or software. These basic mechanisms are the following:

Hit/miss check (processor-side): Decide whether a particular reference can be satisfied locally (*e.g.,* whether or not it hits in the cache).

Request send (processor-side): React to the case where a reference cannot be satisfied locally (*e.g.,* send a message to another processor requesting a copy of the relevant data item and wait for the eventual reply).

Memory-side: Receive a request from another processor, perform any necessary coherence actions, and send a response.

Using these three characteristics, we obtain the following breakdown of the spectrum of DSM implementation techniques that have been discussed in the literature.

All-Hardware In all-hardware DSM systems, all three of these mechanisms are implemented in specialized hardware; the Stanford DASH multiprocessor [27] is a typical all-hardware system.

Mostly Hardware As discussed in Section 4, Alewife implements a mostly hardware DSM system—the processor-side mechanisms are always implemented in hardware, but memory-side support is handled in software when widespread sharing is detected [9]. Dir_1SW and its variations [18, 45] are also mostly hardware schemes.

The Stanford FLASH multiprocessor [25] and Wisconsin Typhoon architecture [33] represent a different kind of mostly hardware DSM system. Both of these systems implement the request send and memory-side functionality in software, but that software is running on a specialized coprocessor associated with every processor/memory pair in the system; only "memory system" code is expected to be run on the coprocessor.

Mostly Software Many software DSM systems are actually mostly software systems in which the "hit/miss check" functionality is implemented in hardware (*e.g.,* by leveraging off of virtual memory protection mechanisms). Typical examples of mostly software systems include Ivy [28], Munin [8], and TreadMarks [14]; coherence units in these systems are the size of virtual memory pages.

Blizzard [37] implements a similar scheme on the CM-5 at the granularity of individual cache lines. By manipulating the error correcting code bits associated with every memory block, Blizzard can control access on a cache-line by cache-line basis.

All-Software In an all-software DSM system, all three of the mechanisms identified above are implemented entirely in software. Several researchers have recently reported on experiences with all-software DSM systems obtained by modifying mostly software DSM systems such that the "hit/miss check" functionality is provided in software [37, 46].

Generally speaking, increased use of software to provide shared-memory functionality tends to decrease application performance because processor cycles spent implementing memory system functionality might otherwise have been spent in application code. However, CRL demonstrates that it is possible to implement all three of these mechanisms in software and still provide performance competitive with hardware implementations on challenging shared-memory applications.

3 The CRL Approach

This section describes the C Region Library (CRL). In terms of the classification presented in the previous section, CRL is an all-software DSM system. Furthermore, CRL is implemented as a library against which user programs are linked; no special hardware, compiler, or operating system support is required.

CRL shares many of the advantages and disadvantages of other software DSM systems when compared to hardware DSMs. In particular, latencies of many communication operations may be significantly higher than similar operations in a hardware-based system. CRL offsets some of this disadvantage by being able to use part of main memory as a large secondary cache instead of relying only on hardware caches, which are typically small because of the cost of the resources required to implement them. In addition, because regions correspond to user-defined data structures, coherence actions transfer exactly the data required by the application. Furthermore, CRL can exploit efficient bulk data transport mechanism when transferring large regions. Finally, because CRL is implemented entirely in software at user level, it is easily modified or extended (*e.g.,* for instrumentation purposes or in order to experiment with different coherence protocols).

3.1 Goals

Several major goals guided the development of CRL. First and foremost, we strove to preserve the essential “feel” of the shared memory programming model without requiring undue limitations on language features or, worse, an entirely new language. In particular, we are interested in preserving the uniform access model for shared data, whether local or remote, that most DSM systems have in common. Second, we were interested in a system that could be implemented efficiently in an all-software context and thus minimized what functionality was required from the underlying hardware and operating system. Systems that take advantage of more complex hardware or operating system functionality (*e.g.,* page-based mostly software DSM systems) are worthy of study, but can suffer a performance penalty because of inefficient interfaces for accessing such features [46]. Finally, we wanted a system that would be amenable to simple and lean implementations in which only a small amount of software overhead sits between applications and access to the message-passing infrastructure used for communication.

3.2 Programming Model

In the CRL programming model, communication is effected through operations on *regions*. Each region is an arbitrarily sized, contiguous area of memory identified by a unique region identifier. New regions can be created dynamically by calling `rgn_create` with one argument, the size of the region to create (in bytes); `rgn_create` returns a region identifier for the newly created region. A region identifier is a portable and stable name for a region (other systems use the term “global pointer” for this concept). Thus `rgn_create` can be thought of as the CRL analogue to `malloc`.

Function	Effect	Argument
`rgn_create`	create a new region	size of region to create
`rgn_delete`	delete a region	region identifier
`rgn_map`	map a region into the local address space	region identifier
`rgn_unmap`	unmap a mapped region	pointer returned by `rgn_map`
`rgn_start_read`	initiate a read operation on a region	pointer returned by `rgn_map`
`rgn_end_read`	terminate a read operation on a region	pointer returned by `rgn_map`
`rgn_start_write`	initiate a write operation on a region	pointer returned by `rgn_map`
`rgn_end_write`	terminate a write operation on a region	pointer returned by `rgn_map`
`rgn_flush`	flush the local copy of a region	pointer returned by `rgn_map`

Table 1: Summary of the CRL interface.

CRL also provides a `rgn_delete` function (analogous to `free`), but in the current implementation, it is a no-op. We plan to implement the `rgn_delete` functionality eventually; doing so should be straightforward, but we haven't found any pressing need to do so for the applications we have implemented to date.

Before accessing a region, a processor must *map* it into the local address space using the `rgn_map` function. `rgn_map` takes one argument, a region identifier, and returns a pointer to the base of the region's data area. A complementary `rgn_unmap` function allows the processor to indicate that it is done accessing the region, at least for the time being. Any number of regions can be mapped simultaneously on a single node, subject to the limitation that each mapping requires at least as much memory as the size of the mapped region, and the total memory usage per node is ultimately limited by the physical resources available. The address at which a particular region is mapped into the local address space may not be the same on all processors. Furthermore, while the mapping is fixed between any `rgn_map` and the corresponding `rgn_unmap`, successive mappings on the same processor may place the region at different locations in the local address space.

Because CRL makes no guarantees about the addresses regions get mapped to, applications that need to store a "pointer" to shared data (*e.g.*, in another region as part of a distributed, shared data structure) must store the corresponding region's unique identifier (as returned by `rgn_create`), *not* the address at which the region is currently mapped. Subsequent references to the data referenced by the region identifier must be preceded by calls to `rgn_map` (to obtain the address at which the region is mapped) and followed by calls to `rgn_unmap` (to clean up the mapping).

After a region has been mapped into the local address space, its data area can be accessed in the same manner as a region of memory referenced by any other pointer: no additional overhead is introduced on a per reference basis. CRL does require, however, that programmers group accesses to a region's data area into *operations* and annotate programs with calls to CRL library functions to delimit them. Two types of operations are available: *read* operations, during which a program is only allowed to read the data area of the region in question, and *write* operations, during which both reads and writes to the data area are allowed. Operations are initiated by calling either `rgn_start_read` or `rgn_start_write`, as appropriate; `rgn_end_read` and `rgn_end_write` are the complementary functions for terminating operations. These functions all take a single argument, the pointer to the base of the region's data area that was returned by `rgn_map` for the region in question. Figure 1 provides a simple example of how these functions might be used in practice.

```
/* compute the dot product of two n-element vectors, each
 * of which is represented by an appropriately sized region
 * x: address at which 1st vector is already mapped
 * y: region identifier for 2nd vector
 */
double dotprod(double *x, rid_t y, int n)
{
  int     i;
  double *z;
  double  rslt;

  /* initiate read operation on 1st vector
   */
  rgn_start_read(x);

  /* map 2nd vector and initiate read operation
   */
  z = (double *) rgn_map(y);
  rgn_start_read(z);

  /* compute dot product
   */
  rslt = 0;
  for (i=0; i<n; i++)
    rslt += x[i] * z[i];

  /* terminate read operations and unmap 2nd vector
   */
  rgn_end_read(x);
  rgn_end_read(z);
  rgn_unmap(z);

  return rslt;
}
```

Figure 1: A simple example of how CRL might be used in practice.

An operation is considered to be *in progress* from the time the initiating `rgn_start_`*op* returns until the corresponding `rgn_end_`*op* is called. Write operations are serialized with respect to all other operations on a region, including those on other processors. Read operations to the same region are allowed to proceed concurrently, independent of the processor on which they are executed. If a newly initiated operation conflicts with those already in progress on the region in question, the call to `rgn_start_`*op* responsible for initiating the operation spins until it can proceed without conflict. The effect of loads from a region's data area when no operation is in progress on that region is undefined; similarly for stores to a region's data area when no write operation is in progress.

In addition to functions for mapping, unmapping, starting operations, and ending operations, CRL provides a *flush* call for mapped regions which causes the local copy of a region to be flushed back to the home node; it is analogous to flushing a cache line in hardware DSM systems. Table 1 summarizes the CRL interface.

Although the programming model provided by CRL is not exactly the same as any "standard" shared memory programming model, our experience is that the programming overhead it causes is minimal. Furthermore, with this modest change to the programming model, CRL implementations are able to amortize the cost of providing the mechanisms described in Section 2 entirely in software over entire operations (typically multiple loads and stores) instead of paying that cost for every reference to potentially shared memory.

The current implementation of CRL supports SPMD-like (single program, multiple data) applications in which a single user thread or process runs on each processor in the system. Interprocessor synchronization can be effected through region operations, barriers, broadcasts, and reductions. Many shared memory applications (*e.g.,* the SPLASH application suites [40]) are written in this style. An initial version of CRL that supports multiple user threads per processor has recently become operational.

3.3 Memory/Coherence Model

The simplest explanation of the coherence model provided by CRL considers entire operations on regions as indivisible units. From this perspective, CRL provides sequential consistency for read and write operations in the same sense that a sequentially consistent hardware-based DSM does for individual loads and stores.

In terms of individual loads and stores, CRL provides a memory/coherence model similar to entry [5] or release consistency [17]. Loads and stores to global data are only allowed within properly synchronized sections (operations), and modifications to a region are only made visible to other processors after the appropriate release operation (a call to `rgn_end_write`). The principal difference between typical implementations of these models and CRL, however, is that synchronization objects (and any association of data with particular synchronization objects that might be necessary) are not provided explicitly by the programmer. Instead, they are implicit in the semantics of the CRL interface: every region has an associated synchronization object (what amounts to a reader-writer lock) which is "acquired" and "released" using calls to `rgn_start_`*op* and `rgn_end_`*op*.

3.4 Prototype Implementation

We have developed an implementation of CRL for Thinking Machines' CM-5 family of multiprocessors and the MIT Alewife machine. In both cases, all communication is effected using active messages [44]. CRL is implemented as a library against which user programs are linked; it is written entirely in

C. Both CM-5 and Alewife versions can be compiled from a single set of sources with conditionally compiled sections to handle machine-specific details (*e.g.,* different message-passing interfaces).

In addition to the basic region functions shown in Table 1, CRL provides a modest selection of global synchronization operations (barrier, broadcast, reductions on double-precision floating-point values). Extending the set of global synchronization operations to make it more complete would be straightforward. On Alewife, CRL implements these operations entirely in software; on the CM-5, we take advantage of the special hardware support for global operations.

3.4.1 Protocol

CRL currently employs a fixed-home, directory-based invalidate protocol similar to that used in many hardware DSM systems. Responses to invalidation messages are always sent back to a region's home node (which collects them and responds appropriately after they have all arrived); the "three-party optimization" in which such responses are sent directly to the requesting node is not used.

In order to handle out-of-order message delivery, a common occurrence when programming with active messages on the CM-5, CRL maintains a 32-bit version number for each region. Each time a remote processor requests a copy of the region, the current version number is recorded in the directory entry allocated for the copy and returned along with the reply message; the current version number is then incremented. By including the version number for a remote copy of a region in all other protocol messages related to that copy, misordered protocol messages are easily identified and either buffered or dropped, as appropriate.

Hardware-based directory protocols typically include provisions for sending a negative acknowledgement (nack) in response to protocol requests that show up at times when it would be inconvenient or difficult to handle them. Upon being nack-ed, requesting nodes are responsible for resending the original request. The overhead of receiving an active message can be significant, even in the most efficient of systems. Thus, employing such an approach in CRL could raise the possibility of livelock situations in which a large number of remote nodes could "gang up" on a home node, saturating it with requests (that always get nack-ed and thus resent) in such a way that forward progress is impeded indefinitely. CRL avoids these problems by never nack-ing protocol requests; those that show up at inconvenient times are simply queued for later processing. Other solutions to this problem are possible (*e.g.,* nack inconvenient requests, but use some amount of backoff before resending them), but we have not investigated them.

3.4.2 Caching

CRL caches both mapped and unmapped regions. First, when an application keeps a region mapped on a particular processor through a sequence of operations, the data associated with the region may be cached between operations. Naturally, the local copy can be invalidated due to other processors initiating operations on the same region. As in hardware DSM systems, whether or not such invalidation actually happens is effectively invisible to the end user (except in terms of any performance penalty it may cause).

Second, whenever a region is unmapped and no other mappings of the region are in progress locally, it is entered into a software table called the *unmapped region cache* (URC); the state of the region's data (*e.g.,* invalid, clean, dirty) is left unchanged. Inserting a region into the URC may require evicting an existing entry. This is accomplished in two steps. First, the region to be evicted is flushed in order to inform its home node that the local copy has been dropped and, if necessary, write back any changes to the data. Second, any memory resources that had been allocated for the evicted region are freed.

Attempts to map remote regions that are not already mapped locally are satisfied from the URC whenever possible. By design, the URC only holds unmapped regions, so any call to `rgn_map` that is satisfied from the URC also causes the region in question to be removed from the URC.

The URC serves two purposes: First, it allows the caching of data (as described above for regions that remain mapped through multiple operations) between different mappings of the same region. If a region with a valid copy of the associated data is placed in the URC (after being unmapped by an application) and the data is not invalidated before the next time the region is mapped, it may be possible to satisfy subsequent calls to `rgn_start_`*op* locally, without requiring communication with the home node. Second, it enables the caching of mappings. Even if the data associated with a region is invalidated while the region sits in the URC (or perhaps was already invalid when the region was inserted into the URC), caching the mapping allows later attempts to map the same region to be satisfied more quickly than they might be otherwise. Calls to `rgn_map` that cannot be satisfied locally require sending a message to the region's home node requesting information (*e.g.,* the size and current version number), waiting for the reply, allocating a local copy for the region, and initializing the protocol metadata appropriately. CRL currently uses a fixed-size URC with 1024 entries.

3.4.3 Status

Our CRL implementation has been operational for several months. We have used it to run a handful of shared-memory-style applications, including two from the SPLASH-2 suite, on a 32-node Alewife system and CM-5 systems with up to 128 processors. A "null" implementation that provides null or identity macros for all CRL functions except `rgn_create` (which is a simple wrapper around malloc) is also available to obtain sequential timings on Alewife, the CM-5, or uniprocessor systems (*e.g.,* desktop workstations).

We plan to make our CRL implementation and CRL versions of several applications available for anonymous ftp sometime during the summer of 1995.

4 Experimental Platforms

This section describes the two platforms that are used for the experiments described in this paper: Thinking Machines' CM-5 family of multiprocessors and the MIT Alewife machine.

4.1 CM-5

The CM-5 [26] is a commercially available message-passing multicomputer with relatively efficient support for low-overhead, fine-grained message passing. The experiments described in this paper were run on a 128-node CM-5 system running version 7.3 Final of the CMOST operating system and version 3.2 of the CMMD message-passing library. Each CM-5 node contains a SPARC v7 processor (running at 32 MHz) and 32 Mbytes of physical memory.

Application codes on the CM-5 can be run with interrupts either enabled or disabled. If interrupts are enabled, active messages arriving at a node are handled immediately by interrupting whatever computation was running on that node; the overhead of receiving active messages in this manner is relatively high. This overhead can be reduced by running with interrupts disabled, in which case incoming active messages simply block until the code running on the node in question explicitly polls the network (or tries to send a message, which implicitly causes the network to be polled). Running with interrupts disabled is not a panacea, however. With interrupt-driven message delivery, the programmer is not aware of when protocol messages are processed by the local node. In contrast, if polling is used, the programmer needs to be aware of when these events happen and to ensure that the network is polled frequently enough to allow those protocol messages to be serviced promptly.

Our CRL implementation for the CM-5 works correctly whether interrupts are enabled or disabled. If it is used with interrupts disabled, users are responsible for ensuring the network is polled frequently enough, as is always the case when programming with interrupts disabled. All CM-5 results presented in this paper are obtained by running with interrupts enabled. It may be possible to obtain somewhat better performance by adding the necessary polling code to applications and running with interrupts disabled; we plan to investigate these issues further in the near future.

In a simple ping-pong test, the round-trip time for four-argument active messages (the size CRL uses for non-data carrying protocol messages) on the CM-5 is approximately 34 microseconds (1088 cycles). This includes the cost of disabling interrupts on the requesting side[1], sending the request, polling until the reply message is received, and then reenabling interrupts. Message delivery on the replying side is interrupt-driven.

For large regions, data-carrying protocol messages use the CMMD's `scopy` functionality to effect data transfer between nodes. `scopy` achieves a transfer rate of 7 to 8 Mbytes/second for large transfers, but because it requires prenegotiation of a special data structure on the receiving node before data transfer can be initiated, performance on small transfers can suffer. To address this problem, CRL employs a special mechanism for data transfers smaller than 384 bytes (the crossover point between the two mechanisms). This mechanism packs three payload words and a destination base address into each four-argument active message; specialized message handlers are used to encode offsets from the destination base address at which the payload words should be stored in the message handler. While this approach cuts the effective transfer bandwidth roughly in half, it provides significantly reduced latencies for small transfers by avoiding the need for prenegotiation with the receiving node.

Networks of workstations with interprocessor communication performance rivaling that of the CM-5 are rapidly becoming reality [6, 30, 41, 43]. For example, Thekkath *et al.* [42] describe the implementation of a specialized data-transfer mechanism implemented on a pair of 25 MHz DECstations connected with a FORE ATM network. They report round-trip times of 45 microseconds (1125 cycles) to read 40 bytes of data from a remote processor. Since this latency is on par with that measured for the CM-5, we expect that the performance of our CRL on the CM-5 is indicative of what should be possible for implementations targeting networks of workstations using current- or next-generation technology.

4.2 Alewife

Alewife [1] is an experimental distributed memory multiprocessor. The basic Alewife architecture consists of processor/memory nodes communicating over a packet-switched interconnection network organized as a low-dimensional mesh (see Figure 2). Each processor/memory node consists of a Sparcle processor [2], an off-the-shelf floating-point unit (FPU), a 64-kilobyte unified instruction/data cache (direct mapped, 16-byte lines), eight megabytes of DRAM, the local portion of the interconnection network, and a Communications and Memory Management Unit (CMMU). Because Sparcle was

[1] Disabling interrupts is required when using `CMAML_rpc` to send an active message; `CMAML_rpc` must be used because CRL's coherence protocol does not fit into the simple request/reply network model that is supported somewhat more efficiently on the CM-5.

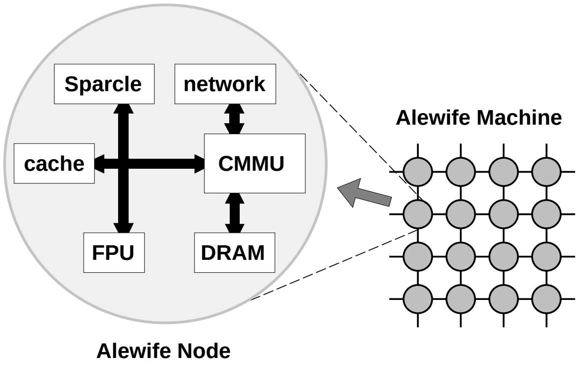

Figure 2: Basic Alewife architecture.

derived from a SPARC v7 processor not unlike that used in the CM-5 nodes, basic processor issues (instruction set, timings, etc.) are quite similar on the two machines.

Alewife provides efficient support for both coherent shared-memory and message-passing communication styles. Shared memory support is provided through an implementation of the LimitLESS cache coherence scheme [9]: limited sharing of memory blocks (up to five remote readers) is supported in hardware; higher-degree sharing is handled by trapping the processor on the home memory node and extending the small hardware directory in software. In general, Alewife's shared memory system performs quite well, enabling speedups comparable to or better than similarly scalable systems.

In addition to providing support for coherent shared memory, Alewife provides the processor with direct access to the interconnection network for sending and receiving messages [24]. Efficient mechanisms are provided for sending and receiving both short (register-to-register) and long (memory-to-memory, block transfer) messages. Using Alewife's message-passing mechanisms, a processor can send a message with just a few user-level instructions. A processor receiving such a message will trap and respond either by rapidly executing a message handler or by queuing the message for later consideration when an appropriate message handler gets scheduled. Scheduling and queuing decisions are made entirely in software.

Two non-fatal bugs in the first-run CMMU silicon warrant mention here. First, because of a timing conflict between the CMMU and the FPU, codes that make significant use of the FPU are limited to running at 20 MHz instead of the target clock rate of 33 MHz. Because of this, all Alewife performance results presented in this paper assume a 20 MHz clock. Second, in order to ensure data integrity when using the block transfer mechanism, it is necessary to flush message buffers from the memory system before sending or initiating storeback on the receiving processor. This overhead cuts the effective peak bandwidth of the block transfer mechanism from approximately 2.2 bytes/cycle (44 Mbytes/second) to roughly 0.9 bytes/cycle (18 Mbytes/second). Both of these bugs will be fixed in a planned CMMU respin.

For the same simple ping-pong test described in Section 4.1, the round-trip time for four-word active messages (using interrupt-driven message delivery on both ends) on Alewife is approximately 14 microseconds (280 cycles). Even in terms of absolute time (without correcting for the differences in clock speed), this is more than a factor of two faster than the CM-5. In our current Alewife CRL implementation, active message latencies are somewhat higher, however, because all protocol message handlers are effectively transitioned into full-fledged threads that can be interrupted by incoming messages. This transition prevents long-running handlers from blocking further message delivery and

		CM-5		Alewife		Alewife (native)	
		cycles	μsec	cycles	μsec	cycles	μsec
start read	hit	80	2.5	47	2.3	—	—
end read		96	3.0	50	2.5	—	—
start read	miss, no invalidations	1763	55.1	580	29.0	38	1.9
start write	miss, one invalidation	3459	108.1	978	48.9	66	3.3
start write	miss, six invalidations	4157	129.9	1935	96.7	707	35.4

Table 2: Measured CRL latencies, 16-byte regions (in both cycles and microseconds). Measurements for Alewife's native shared memory system are provided for comparison.

		CM-5		Alewife	
		cycles	μsec	cycles	μsec
start read	miss, no invalidations	3581	111.9	642	32.1
start write	miss, one invalidation	5312	166.0	1046	52.3
start write	miss, six invalidations	6006	187.7	2004	100.2

Table 3: Measured CRL latencies, 256-byte regions (in both cycles and microseconds).

causing network congestion. Currently, this transition adds approximately 248 cycles (12.4 microseconds) to the round-trip time, but minor functionality extensions planned for the CMMU respin will make it possible to reduce this overhead by at least an order of magnitude.

A sixteen-node Alewife machine has been operational since June, 1994; this system was expanded to 32 nodes in November, 1994. Larger systems will become available over the course of the next year.

5 Results

This section presents performance results for CRL. These results are described in two subsections: the first discusses the latencies of various basic events as measured with a simple microbenchmark; the second presents results from three applications (blocked LU, Water, and Barnes-Hut). In all cases, results are presented not only for the two CRL implementations (CM-5 and Alewife), but also for Alewife's native shared memory support.

The basic latencies for Alewife CRL presented in Section 5.1 do not include the overhead of flushing message buffers and transitioning message handlers into threads as discussed in Section 4; these figures were obtained using a version of the Alewife CRL implementation in which code related to flushing message buffers and transitioning handlers into threads had been removed. Thus, these latencies are indicative of what will be possible after the Alewife CMMU respin. In the existing Alewife CRL implementation, these overheads cause applications to see latencies between 55 to 75 percent larger than those shown here. All other Alewife CRL results reported in this paper include this additional overhead and are thus somewhat worse than what should be possible after the respin.

5.1 Basic Latencies

The following simple microbenchmark is used to measure the cost of various CRL events. Some number of regions are allocated on a selected home node (the measurements presented here were taken with 64 regions). Situations corresponding to desired events (*e.g.,* a start write on a remote node that requires other remote read copies be invalidated) are constructed mechanically for some subset of the regions; the time it takes yet another processor to execute a simple loop calling the relevant CRL function for each of these regions is then measured. We compute the time for the event in question by repeating this process for all numbers of regions between one and the number allocated and computing the linear regression of the number of regions against measured times; the slope thus obtained is taken to be the time per event.

Invocations of `rgn_map` that can be satisfied locally (*e.g.,* because the call was made on the home node for the region in question, the region is already mapped, or the region is present in the URC) are termed "hits." On both Alewife and the CM-5, invocations of `rgn_map` that are hits cost between 80 and 170 cycles, depending on whether or not the region in question had to be removed from the unmapped region cache. Calls to `rgn_map` that cannot be satisfied locally ("misses") are more expensive (roughly 820 cycles on Alewife and 2300 cycles on the CM-5). This increase reflects the cost of sending a message to the region's home node, waiting for a reply, allocating a local copy for the region, and initializing the protocol metadata appropriately. Invocations of `rgn_unmap` take between 30 and 100 cycles; the longer times corresponding to the cases in which the region being unmapped needs to be inserted into the unmapped region cache.

Table 2 shows the measured latencies for a number of typical CRL events, assuming 16-byte regions. The first two lines ("start read, hit" and "end read") represent events that can be satisfied entirely locally. The other lines in the table show miss latencies for three situations: "start read, miss, no invalidations" represents a simple read miss to a remote location requiring no other protocol actions; "start write, miss, one invalidation" represents a write miss to a remote location that also requires a read copy of the data on a third node to be invalidated; "start write, miss, six invalidations" represents a similar situation in which read copies on six other nodes must be invalidated.

Latencies for Alewife's native shared memory system are provided for comparison. The first two cases shown here (read miss, no invalidations, and write miss, one invalidation) are situations in which the miss is satisfied entirely in hardware. The third case (write miss, six invalidations) is one in which LimitLESS software must be invoked, because Alewife only provides hardware support for up to five outstanding copies of a cache line. Note that for 16-byte regions (the same size as the cache lines used in Alewife), the CRL latencies are roughly a factor of 15 larger than those for a request handled entirely in hardware; this factor is entirely due to time spent executing CRL code and the overhead of active message delivery.

Table 3 shows how the miss latencies given in Table 2 change when the region size is increased to 256 bytes. Note that for Alewife, these latencies are only 60 to 70 cycles larger than those for 16-byte regions; the fact that the differences are so small is a testament to the efficiency of Alewife's block transfer mechanism.

Interestingly, these latencies indicate that with regions of a few hundred bytes in size, CRL achieves a bandwidth similar to that provided by hardware-supported shared memory (a 16-byte cache line @ 1.9 microseconds = 8.4 Mbytes/second; a 256-byte region @ 32.1 microseconds = 8.0 Mbytes/second). While such a simple calculation ignores numerous important issues, it does provide a rough indication of the data granularity that CRL should be able to support efficiently when built on top of fast message-passing mechanisms. Note that because the CM-5 provides less efficient mechanisms for bulk data

	Blocked LU			Water			Barnes-Hut		
	CM-5	Alewife		CM-5	Alewife		CM-5	Alewife	
	CRL	CRL	SM	CRL	CRL	SM	CRL	CRL	SM
sequential	24.89	52.05	52.05	10.63	22.41	22.41	11.67	22.83	22.83
1 proc	25.35	53.56	52.15	12.21	23.85	22.49	23.07	34.97	22.96
32 procs	1.62	2.36	2.62	1.33	1.19	1.01	1.70	1.90	1.36

Table 4: Application running times (in seconds).

		Blocked LU		Water		Barnes-Hut	
		CM-5	Alewife	CM-5	Alewife	CM-5	Alewife
CRL, 1 proc	map count (x 1000)	84.58	84.58	—	—	983.60	983.60
	op count (x 1000)	84.63	84.63	269.32	269.32	992.22	992.22
CRL, 32 procs	map count (x 1000)	2.81	2.81	—	—	30.76	30.76
	(miss rate, %)	15.30	15.30	—	—	1.27	1.25
	op count (x 1000)	2.78	2.78	8.68	8.68	31.27	31.14
	(miss rate, %)	14.29	14.29	7.84	9.23	4.73	4.77
	msg count (x 1000)	1.65	1.65	2.56	3.03	5.89	5.90

Table 5: Application characteristics when running under CRL (measurements are average values expressed on a per-processor basis.)

transfer, the CM-5 implementation of CRL requires much larger regions to provide effective remote data access bandwidth approaching that delivered by Alewife's hardware-supported shared memory.

5.2 Application Performance

While comparisons of the performance of low-level mechanisms can be revealing, end-to-end performance comparisons of real applications are far more important. The remainder of this section presents results obtained from running three applications (Blocked LU, Water, and Barnes-Hut) on top of both CRL implementations and Alewife's native shared memory support.

For all of the applications discussed in this paper, judicious use of conditional compilation allows the same set of sources to be compiled to use either CRL (on either Alewife or the CM-5) or shared memory (Alewife only) to effect interprocessor communication. The shared-memory versions of applications use the hardware-supported shared memory directly and contain no calls to the CRL functions described in Section 3 (for brevity's sake, the rest of the paper uses the term "Alewife SM" to refer to this case). None of the applications employ any prefetching.

Table 4 summarizes the running times for the sequential, CRL, and shared memory (SM) versions of the three applications. Sequential running time is obtained by compiling each application against the null CRL implementation described in Section 3 and running on a single node of the architecture in question; this time is used as the basepoint for computing application speedup. Note that the running times for the CRL versions of applications running on one processor are larger than the sequential running times. This difference represents the overhead of calls to CRL functions—even CRL calls that

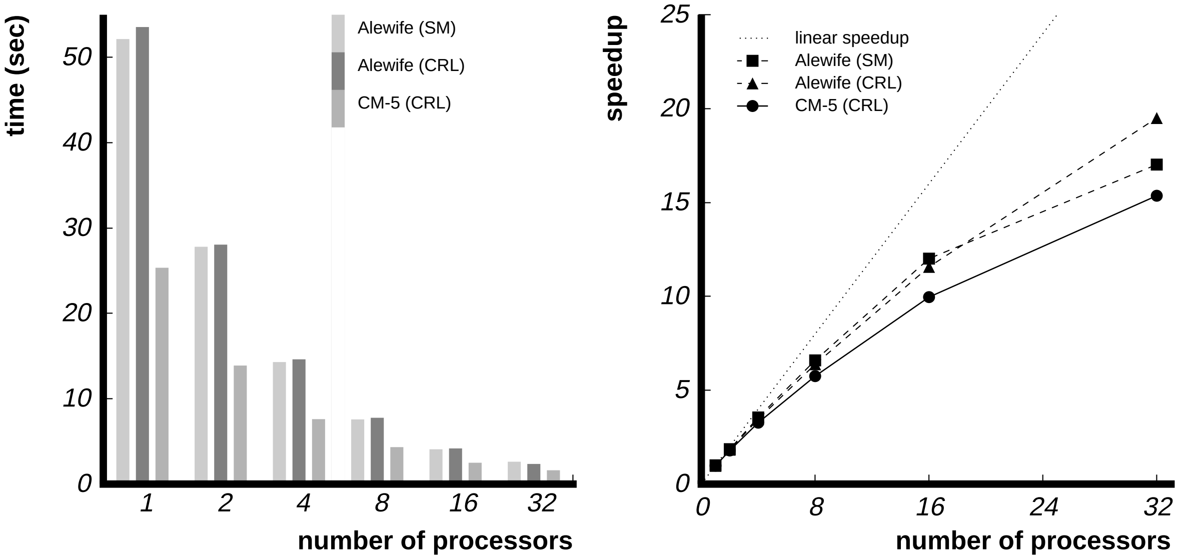

Figure 3: Absolute running time (left) and speedup (right) for Blocked LU (500x500 matrix, 10x10 blocks).

"hit" incur overhead, unlike hardware systems where hits (*e.g.*, in a cache) incur no overhead.

Table 5 presents event counts obtained by compiling each application against an instrumented version of the CRL library and running the resulting binary. The instrumented version of the CRL library collected many more statistics than those shown here; applications linked against it run approximately 10% slower than when linked against the unmodified library. Table 5 shows counts for three different events: "map count" indicates the number of times regions were mapped (because calls to `rgn_map` and `rgn_unmap` are always paired, this number also represents the number of times regions were unmapped); "op count" indicates the total number of operations executed (paired calls to `rgn_start_`*op* and `rgn_end_`*op*); "msg count" shows the number of protocol messages sent and received during the time in question. For the 32 processor results, miss rates are also shown; these rates indicate the fraction of calls to `rgn_map` and `rgn_start_`*op* that cannot be satisfied locally (without requiring interprocessor communication). All counts are average figures expressed on a per-processor basis.

Map count and miss rates for Water are shown as '—' because the application's entire data set is kept mapped on all nodes at all times; regions are mapped once at program start time and never unmapped. While this may not be a good idea in general, it is reasonable for Water because the data set is relatively small (a few hundred kilobytes) and is likely to remain manageable even for larger problem sizes.

5.2.1 Blocked LU

Blocked LU implements LU factorization of a dense matrix; the version reported on here is based on one described by Rothberg *et al.* [34]. In the CRL version of the code, a region is created for each block of the matrix to be factored. The results presented here are for a 500x500 matrix using 10x10 blocks; thus the size of each region is 800 bytes.

Figure 3 shows the performance of the three different versions of Blocked LU (CM-5 CRL, Alewife CRL, Alewife SM) on up to 32 processors. The left-hand plot shows absolute running time, without correcting for differences in clock speed between the CM-5 (32 MHz) and Alewife (20 MHz). The

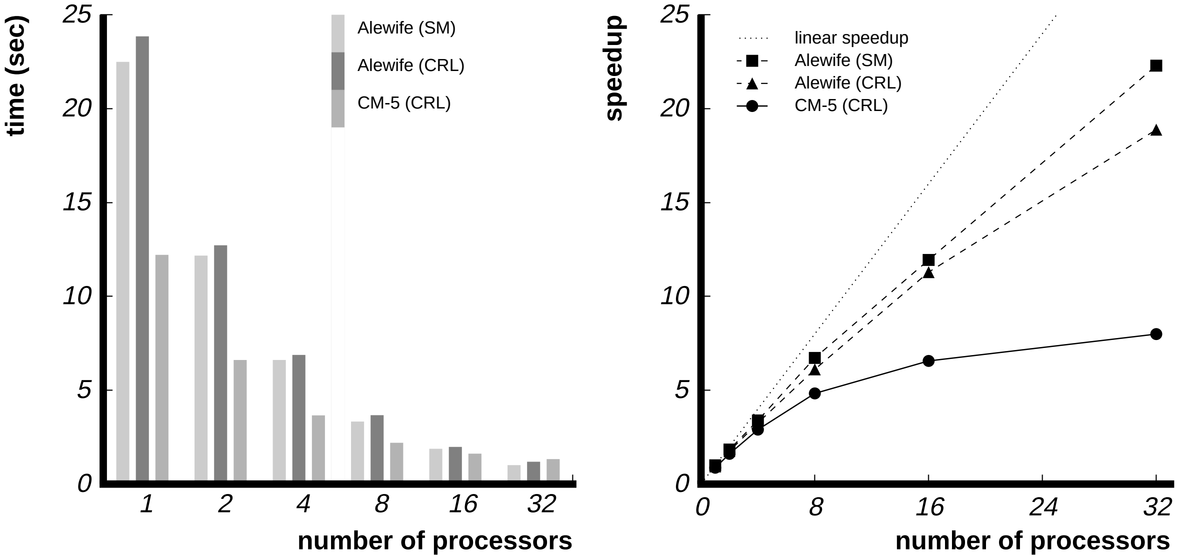

Figure 4: Absolute running time (left) and speedup (right) for Water (512 molecules).

right-hand plot shows speedup; the basepoints for the speedup calculations are the sequential running times shown in Table 4 (thus both Alewife curves are normalized to the same basepoint, but the CM-5 speedup curve uses a different basepoint).

All three implementations perform well, delivering speedups of between 15 and 20 on 32 processors. This is not particularly surprising; it is relatively well known that Blocked LU decomposition does not present a particular challenging communication workload. The data in Tables 4 and 5 confirm this belief, indicating acceptable miss rates and large granularity: an average of roughly 9,400 (CM-5) and 12,300 (Alewife) cycles of useful work per operation. (This figure is obtained by dividing the sequential running time by the number of operations executed by the CRL version of the application running on one processor.)

Somewhat surprising, however, is the fact that Alewife CRL outperforms Alewife SM by roughly 15% on 32 processors. This occurs because of LimitLESS software overhead. On 16 processors, only a small portion of the LU data set is shared more widely than the five-way sharing supported in hardware, so LimitLESS software is only invoked infrequently. On 32 processors, this is no longer true: over half of the data set is read by more than five processors at some point during program execution. The overhead incurred by servicing some portion of these requests in software allows the performance of Alewife CRL to outstrip that of Alewife SM.

5.2.2 Water

The Water application used in this study is the "n-squared" version from the SPLASH-2 application suite; it is a molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. Applications like Water are typically run for tens or hundreds of iterations (time steps), so the time per iteration in the "steady state" dominates any startup effects. Therefore, to determine running time, we run the application for three iterations and take the average of the second and third iteration times (thus eliminating timing variations due to startup transients that occur during the first iteration).

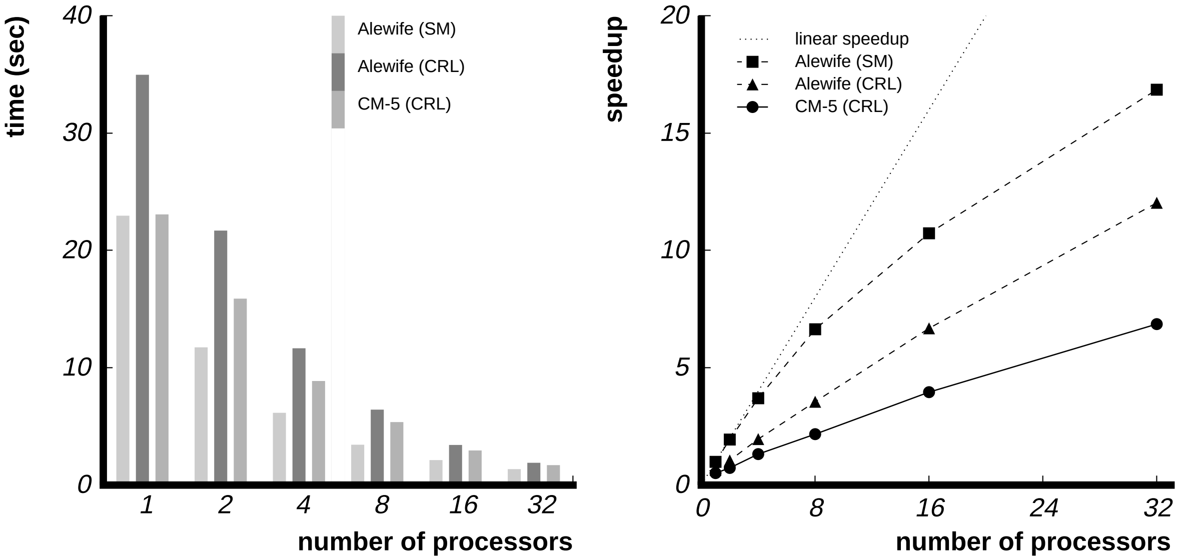

Figure 5: Absolute running time (left) and speedup (right) for Barnes-Hut (4096 bodies).

In the CRL version of the code, a region is created for each molecule data structure; the size of each such region is 672 bytes. Three small regions (8, 24, and 24 bytes) are also created to hold several running sums that are updated every iteration by each processor. The results presented here are for a problem size of 512 molecules.

Figure 4 shows the absolute running time and speedup curves for the three versions of Water. Both Alewife SM and Alewife CRL perform quite well (speedups of 22.3 and 18.9 on 32 processors, respectively), but the performance of CM-5 CRL on this application is somewhat disappointing (a speedup of 8.0 on 32 processors).

Examining the granularity of Water helps explain the disappointing speedup for CM-5 CRL. The data in Tables 4 and 5 indicate granularities of approximately 1,260 and 1,660 cycles of useful work per operation, nearly a factor of eight smaller than those for Blocked LU. Even given a relatively low miss rate (7.84%), this granularity is small enough that the larger miss latencies for CM-5 CRL begin to contribute a significant portion of the total running time, thus limiting the possible speedup. In contrast, Alewife's more efficient communication mechanisms allow Alewife CRL to support this granularity with only a modest performance hit (less than 20% worse than Alewife SM).

In spite of the fact that CRL is an all-software DSM system, it performs comparably with existing mostly software DSM systems. The CM-5 CRL speedup for Water (4.8 on eight processors) is slightly better than that reported for TreadMarks [14], a second-generation page-based mostly software DSM system (4.0 on eight processors, the largest configuration results have been reported for)[2].

5.2.3 Barnes-Hut

The Barnes-Hut application is also taken from the SPLASH-2 application suite; it employs hierarchical n-body techniques to simulate the evolution of a system of bodies under the influence of gravitational forces. As was the case with Water, applications like Barnes-Hut are often run for a large number of

[2] The SPLASH-2 version of Water used in this paper incorporates the "M-Water" modifications suggested by Cox *et al.* [14].

iterations, so the steady-state time per iteration is an appropriate measure of running time. Since the startup transients in Barnes-Hut persist through the first two iterations, we determine running time by running the application for four iterations and taking the average of the third and fourth iteration times.

In the CRL version of the code, a region is created for each of the octtree data structure elements in the original code: bodies (108 bytes), tree cells (88 bytes), and tree leaves (100 bytes). In addition, all versions of the code were modified to use the reduction functionality provided by CRL for computing global sums, minima, and maxima. The results presented here are for a problem size of 4,096 bodies (one-quarter of the suggested base problem size). Other application parameters (Δt and θ) are scaled appropriately for the smaller problem size [40].

Barnes-Hut represents a challenging communication workload. First, communication is relatively fine-grained, both in terms of region size (roughly 100 bytes) and the potential computation-to-communication ratio: the granularity of useful work per operation—376 and 460 cycles on the CM-5 and Alewife, respectively—is roughly a factor of 3.5 smaller than for Water. Second, while Barnes-Hut exhibits a reasonable amount of temporal locality, access patterns are quite irregular due to large amounts of "pointer chasing" through the data structure around which Barnes-Hut is built. In fact, Barnes-Hut and related hierarchical n-body methods present a challenging enough communication workload that they have been used by some authors as the basis of an argument in favor of aggressive hardware support for cache-coherent shared memory [38, 39].

Figure 5 shows the absolute running time and speedup curves for the three versions of Barnes-Hut. Once again, Alewife SM delivers the best performance, achieving a speedup of 16.9 on 32 processors, but Alewife CRL is not far behind with a speedup of 12.0. Thus, while Alewife's aggressive hardware support for coherent shared memory does provide some performance benefit, the reduction in running time over Alewife CRL's all-software approach is somewhat less than one might expect (roughly 30%; other experiments indicate that this gap decreases slightly for larger problem sizes).

The Barnes-Hut speedups for CM-5 CRL are somewhat smaller (6.9 on 32 processors). As was the case with Water, this is primarily due to the small granularity of useful work per operation (376 cycles); this granularity is small enough that even in the face of fairly low operation miss rates (4.73%), the larger miss latencies for CM-5 CRL cause significant performance degradation.

Because larger problem sizes lead to decreased miss rates for Barnes-Hut, such performance problems tend to decrease with larger problem sizes. Figure 6 demonstrates this fact by plotting the performance of the CM-5 CRL version of Barnes-Hut on up to 128 processors for both the 4,096 body problem size discussed above and the suggested problem size of 16,384 bodies. For the larger machine sizes (64, 96, and 128 processors), the increased problem size enables speedups between a factor of 1.5 and 2.0 better than those for 4,096 bodies.

5.3 Summary of Results

We draw two major conclusions from the results presented in this section. First, the CRL implementation on Alewife demonstrates that an all-software DSM system can achieve application performance competitive with hardware-supported DSM systems. For Barnes-Hut with 4,096 particles, the speedup on 32 processors using Alewife CRL is within 30% of that for Alewife SM. As discussed above, we expect this gap to narrow somewhat after the Alewife CMMU respin.

Second, our measurements drive home the point that messaging substrates must provide both low latency and high bandwidth. In particular, communication performance on the CM-5 is not sufficient for CRL to be competitive with hardware-supported DSMs. Communication performance closer to that provided by Alewife is necessary for an implementation of CRL targeting networks of workstations to

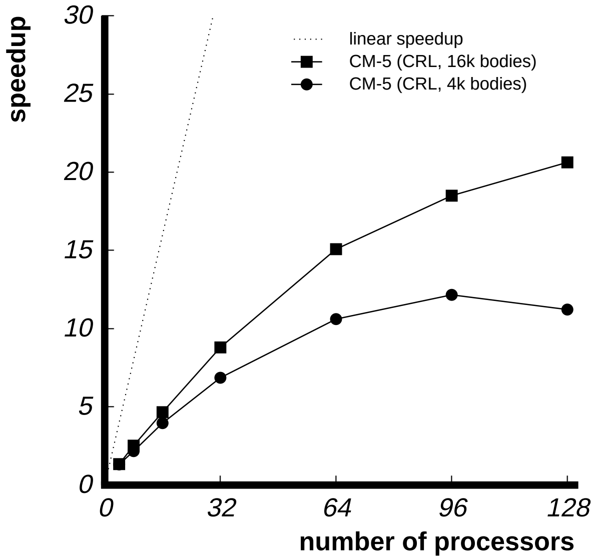

Figure 6: Barnes-Hut performance for larger problems (16,384 bodies) and machine sizes (128-node CM-5).

be competitive with hardware systems.

6 Discussion and Future Work

Section 3.1 described three goals that guided the development of CRL; we believe that our current CRL implementations meet these goals. Our experience porting several applications to CRL and judiciously inserting preprocessor directives so the same sources can be compiled for use with either CRL or shared memory confirm that CRL preserves the essential "feel" of shared memory. Our implementation meets the all-software criteria; CRL should port easily to other systems with support for an active message communication style. Finally, the performance results of the previous section validate the notion that CRL is amenable to simple and lean implementations where the amount of software overhead between applications and the message-passing infrastructure is kept to a minimum.

CRL requires programmers to insert library calls to delimit operations on regions. This modest diversion from "standard" shared memory programming models enables CRL implementations to amortize the cost of providing the mechanisms described in Section 2 over entire operations (typically multiple loads and stores) instead of incurring comparable overhead on every reference to potentially shared memory. Annotations similar to those required by CRL are necessary in aggressive hardware DSM implementations (*e.g.,* those providing release consistency) when writing to shared data. CRL requires such annotations whether reading or writing shared data, similar to entry consistency [5]. Based on our experience with the applications described in this paper, we feel that the additional programming overhead of doing so is minimal. Therefore, we believe CRL is an effective approach to providing a distributed shared memory abstraction.

Calls to CRL library functions also provide a performance advantage. For example, calls that initiate operations provide "prefetch" information that is not available to a hardware-based DSM. We plan to investigate this issue by inserting prefetch instructions into the shared memory versions of applications and measuring the changes in execution time; we expect any improvements to be relatively modest.

Several promising research directions follow from work described in this paper. First, we plan to investigate whether there is any performance benefit to a mostly software CRL implementation that leverages off of virtual memory protection mechanisms in a manner similar to page-based mostly software DSM systems. We believe that the value of doing so will probably be greatly enhanced by operating system structures that allow low-overhead, flexible access to these mechanisms from user level. Second, the results presented in this paper reiterate the need for network interfaces that provide not only low latency but also high bandwidth. Any furthering of the state of the art in that domain would not only help CRL but likely have broad impact across the spectrum of distributed systems.

7 Related Work

Except for the notion of mapping and unmapping regions, the programming interface CRL presents to the end user is similar to that provided by Shared Regions [35]; the same basic notion of synchronized access ("operations") to regions ("objects") also exists in other DSM programming systems [3, 11]. The Shared Regions work arrived at this interface from a different set of constraints; their goal was to provide software coherence mechanisms on machines that support non-cache-coherent shared memory in hardware. CRL could be provided on such systems using the same implementation techniques and defining `rgn_map` and `rgn_unmap` to be null macros.

A number of other approaches to providing coherence in software on top of non-cache-coherent shared-memory hardware have also been explored [15, 22]. Like the Shared Regions work, these research efforts differ from that described in this paper both in the type of hardware platform targeted (non-cache-coherent shared memory *vs.* message passing) and the use of simulation to obtain controlled comparisons with cache-coherent hardware DSM (when such a comparison is provided).

Chandra *et al.* [10] propose a hybrid DSM protocol in which annotations similar to those described in this paper are used to demark access to regions of shared data. Coherence for regions annotated thusly is provided using software DSM techniques analogous to those used by CRL; hardware DSM mechanisms are used for coherence on all other memory references. All synchronization must be effected through hardware DSM mechanisms. In contrast, CRL is an all-software DSM system in which *all* communication and synchronization is implemented using software DSM techniques.

Several all-software DSM systems that employ an object-based approach have been developed (*e.g.*, Amber [13], Orca [3]). Like CRL, these systems effect coherence at the level of application-defined regions of memory ("objects"). Any necessary synchronization, data replication, or thread migration functionality is provided automatically at the entry and exit of methods on shared objects. Existing systems of this type either require the use of an entirely new object-oriented language [3, 20] or only allow the use of a subset of an existing one [13]. In contrast, CRL is not language specific; the basic CRL interface could easily be provided in any imperative programming language.

Scales and Lam [36] have described SAM, a shared object system for distributed memory machines. SAM is based on a new set of primitives that are motivated by optimizations commonly used on distributed memory machines. These primitives are significantly different than "standard" shared memory models. Like SAM, CRL is implemented as a portable C library. Both CRL and SAM achieve good performance on distributed memory machines.

Midway is a software DSM system based on entry consistency [5]. Both mostly software and all-software version of Midway have been implemented [46]. CRL differs from Midway in provided a simpler programming model that bundles synchronization and data access. To the best of our knowledge, Midway has only been implemented on a small cluster of workstations connected with an ATM network.

Cid [31], like CRL, is an all-software DSM system in which coherence is effected on regions ("global objects") according to source code annotations provided by the programmer. Cid differs from the current CRL implementation in its potentially richer support for multithreading, automatic data placement, and load balancing. To date, Cid has only been implemented and run on a small cluster of workstations connected by FDDI [32]. Unlike Midway and Cid, CRL runs on two large-scale platforms and has been shown to deliver performance competitive with hardware DSM systems.

Several researchers have reported results comparing the performance of systems at adjacent levels of the classification presented in Section 2 (*e.g.*, all-hardware vs. mostly hardware [9, 18, 45], mostly software vs. all-software [37, 46]), but to our knowledge, only Cox *et al.* [14] have published results from a relatively controlled comparison of hardware and software DSM systems. While their experiments kept many factors fixed (*e.g.*, processor, caches, compiler), they were unable to keep the communication substrate fixed: they compare a bus-based, all-hardware DSM system with a mostly software DSM system running on a network of workstations connected through an ATM switch. Furthermore, their results for systems with more than eight processors were acquired through simulation. In contrast, the results presented in this paper were obtained through controlled experiments in which only the communication interfaces used by the programming systems were changed. Experimental results comparing hardware and software DSM performance are shown for up to 32 processors (Alewife); software DSM results are shown for up to 128 processors (CM-5).

Klaiber and Levy [21] describe a set of experiments in which data-parallel (C*) applications are compiled such that all interprocessor communication is provided through a very simple library interface. They employ a simulation-based approach to study the message traffic induced by the applications given implementations of this library for three broad classes of multiprocessors: message passing, non-coherent shared memory, and coherent shared memory. In contrast, this paper shows results comparing the absolute performance of implementations of CRL for two message-passing platforms and compares the delivered application performance to that achieved by a hardware-supported DSM.

In terms of comparing message passing and shared memory, most other previous work has either compared the performance of applications written and tuned specifically for each programming model [8, 12] or looked at the performance gains made possible by augmenting a hardware DSM system with message passing primitives [23]. Such research addresses a different set of issues than those discussed in this paper, which takes a shared memory programming model as a given and provides a controlled comparison of hardware and software implementations of distributed shared memory.

Schoinas *et al.* [37] describe a taxonomy of shared-memory systems that is similar in spirit to that provided in Section 2. Their scheme differs from that in Section 2 in its focus on processor-side actions and emphasis of specific implementation techniques instead of general mechanisms.

8 Conclusions

This paper introduces CRL, a new all-software region-based DSM system. Even though it requires no special compiler, hardware, or operating system support beyond the ability to send and receive messages, CRL provides a simple, portable shared address space programming model that is capable of delivering good performance on a wide range of multiprocessor and distributed system architectures.

The paper describes CRL implementations for two platforms, the CM-5 and the MIT Alewife machine, and presents results for up to 128 processors on the CM-5 and up to 32 processors on Alewife. We demonstrate that CRL performs well on the CM-5, delivering speedups comparable to other state-of-the-art software DSM systems. Furthermore, we show that the Alewife implementation of CRL delivers speedups within 30% of those provided by hardware-supported DSMs, even for challenging applications (Barnes-Hut) and small problem sizes.

These results suggest that the CRL approach can be used to provide an efficient distributed shared memory programming environment without any need for hardware support beyond that required for high-performance message-based communication.

9 Acknowledgements

The research was supported in part by ONR contract # N00014-94-1-0985, in part by NSF Experimental Systems grant # MIP-9012773, and in part by an NSF Young Investigator Award.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995. To appear.

[2] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An Evolutionary Processor Design for Multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[3] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.

[4] Henri E. Bal and M. Frans Kaashoek. Object Distribution in Orca using Compile-Time and Run-Time Techniques. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 162–177, September 1993.

[5] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON'93)*, pages 528–537, February 1993.

[6] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, pages 29–36, February 1995.

[7] Martin C. Carlisle, Anne Rogers, John H. Reppy, and Laurie J. Hendren. Early Experiences with Olden. In *Conference Record of the Sixth Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[8] John B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice University, August 1993.

[9] David L. Chaiken and Anant Agarwal. Software-Extended Coherent Shared Memory: Performance and Cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 314–324, April 1994.

[10] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proceedings of the Eighth International Conference on Supercomputing*, pages 274–288, July 1994.

[11] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data Locality and Load Balancing in COOL. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, pages 249–259, May 1993.

[12] 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*, pages 61–73, October 1994.

[13] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 147–158, December 1989.

[14] 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.

[15] Alan L. Cox and Robert J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[16] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[17] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990.

[18] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[19] Wilson C. Hsieh, Paul Wang, and William E. Weihl. Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems. In *Proceedings of the Fourth Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 239–248, May 1993.

[20] Vijay Karamcheti and Andrew Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of Supercomputing '93*, November 1993.

[21] Alexander C. Klaiber and Henry M. Levy. A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 94–105, April 1994.

[22] Leonidas I. Kontothanassis and Michael L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the First Symposium on High-Performance Computer Architecture*, pages 286–295, January 1995.

[23] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[24] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the International Conference on Supercomputing*, pages 195–206, July 1993.

[25] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[26] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[27] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[28] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Computing*, pages 94–101, 1988.

[29] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[30] Ron Minnich, Dan Burns, and Frank Hady. The Memory-Integrated Network Interface. *IEEE Micro*, pages 11–20, February 1995.

[31] Rishiyur S. Nikhil. Cid: A Parallel, "Shared-memory" C for Distributed-Memory Machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[32] Rishiyur S. Nikhil. Personal communication, March 1995.

[33] Steve 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 325–336, April 1994.

[34] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 14–25, May 1993.

[35] Harjinder S. Sandhu, Benjamin Gamsa, and Songnian Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proceedings of the Fourth Symposium on Principles and Practices of Parallel Programming*, pages 229–238, May 1993.

[36] Daniel J. Scales and Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 101–114, November 1994.

[37] 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*, pages 297–306, October 1994.

[38] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer*, pages 45–55, July 1994.

[39] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of Hierarchical N-body Methods for Multiprocessor Architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

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

[41] Chandramohan A. Thekkath and Henry M. Levy. Limits to Low-Latency Communication on High-Speed Networks. *Transactions on Computer Systems*, pages 179–203, May 1993.

[42] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1994.

[43] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, pages 46–53, February 1995.

[44] Thorsten von Eicken, David Culler, Seth Goldstein, and Klaus Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[45] 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 156–167, May 1993.

[46] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, November 1994.