CRL: High-Performance All-Software Distributed Shared Memory

Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach
MIT Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.
{tuna, kaashoek, kerr}@lcs.mit.edu

Abstract

The *C Region Library* (CRL) is 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, region-based shared address space programming model that is capable of delivering good performance on a wide range of multiprocessor and distributed system architectures. Each region is an arbitrarily sized, contiguous area of memory. The programmer defines regions and delimits accesses to them using annotations.

We have developed CRL implementations for two platforms: the Thinking Machines 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 15% of those provided by Alewife's native support for shared memory, even for challenging applications (*e.g.,* Barnes-Hut) and small problem sizes.

1 Introduction

The *C Region Library* (CRL) is a new all-software DSM system for message-passing multicomputers and distributed systems. The challenge in building such a 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. The programmer defines regions and delimits accesses to them using annotations. 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 fixed-size unit, CRL avoids the concomitant problems of false sharing (when coherence units are too large) or inefficient use of bandwidth (when coherence units are too small). We show that CRL is capable of delivering performance competitive with hardware-supported DSM systems.

This research was supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985, by an NSF National Young Investigator Award, by Project Scout under ARPA contract MDA972-92-J-1032, and by a fellowship from the Computer Measurement Group.

Copyright © 1995 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that new copies bear this notice and the full citation on the first page. Copyrights for components of this WORK owned by others than ACM must be honored. Abstracting with credit is permitted.

To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request Permissions from Publications Dept, ACM Inc., Fax +1 (212) 869-0481, or <permissions@acm.org>.

The work described in this paper builds upon a large body of research into the construction of software DSM systems. Three key features distinguish CRL from other software DSM systems. First, CRL is system- and language-independent. Our current implementation provides a C language interface; providing the CRL interface 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, compiler, or operating system beyond that necessary to send and receive messages. Third, CRL is efficient. Little 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 to provide all three in a simple, coherent package. Section 7 discusses in detail how CRL relates to other software DSM systems.

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 [44, 45].

In spite of this fact, message passing environments such as PVM (*Parallel Virtual Machine*) [19] and MPI (*Message Passing Interface*) [35] 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, 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 to deliver 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 [29], the perfor-

Appears in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

mance 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. We find that CRL achieves speedups within 15% 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 three basic mechanisms. Section 3 describes the goals, programming model, and implementation of CRL. Section 4 provides details about the experimental platforms used in this research. 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 are also possible (*e.g.,* those in which threads of computation are migrated to the data they reference [4, 7, 14, 23]).

We classify systems by three basic mechanisms required to implement DSM and whether 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 [33] and KSR-1 [26] are typical all-hardware systems.

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 [10]. Dir_1SW and its variations [21, 53] are also mostly hardware schemes.

The Stanford FLASH multiprocessor [31] and Wisconsin Typhoon architecture [39] 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 [34], Munin [8], and TreadMarks [25]; coherence units in these systems are the size of virtual memory pages.

Blizzard [43] 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 (*e.g.,* Orca [3]). 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 [43, 54].

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, the latencies of many communication operations may be significantly higher than similar operations in a hardware-based system. Four properties of CRL allow it to offset some of this disadvantage. First, CRL is 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. Second, if regions are chosen to correspond to user-defined data structures, coherence actions transfer exactly the data required by the application. Third, CRL can exploit efficient bulk data transport mechanisms 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 were 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 the functionality 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) can suffer a performance penalty because of inefficient interfaces for accessing such features [54]. 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 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`.

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 loads and stores 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: region identifier for 1st vector
 * y: address at which 2nd vector is already mapped
 */
double dotprod(rid_t x, double *y, int n)
{
  int     i;
  double *z;
  double  rslt;

  /* map 1st vector and initiate read operation */
  z = (double *) rgn_map(x);
  rgn_start_read(z);

  /* initiate read operation on 2nd vector */
  rgn_start_read(y);

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

  /* terminate read operations and unmap 1st vector */
  rgn_end_read(y);
  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 the same 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 invocation of 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 that causes the local copy of a region to be flushed back to the home node. By selectively flushing regions, it may be possible to reduce future coherence traffic (*e.g.,* invalidations) related to the flushed regions. Flushing a region 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 quite small. 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.

This paper describes an implementation of CRL that supports SPMD-like (single program, multiple data) applications in which

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.

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 [46, 52]) are written in this style. Although an experimental version of CRL that supports multiple user threads per processor and migration of threads between processors is operational [22], all results reported in this paper were obtained using the single-threaded version.

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 [20]. Loads and stores to global data are allowed only 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 [50]. 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 using message passing. On the CM-5, CRL takes advantage of the special hardware support for global operations. For the applications used in this study, we find that employing a software-based implementation of these mechanisms typically changes running time by no more than a few percent (sometimes a slight increase, more often a slight decrease).

3.4.1 Protocol

CRL currently employs a fixed-home, directory-based invalidate protocol similar to that used in many hardware DSM systems (*e.g.,* Alewife, DASH). 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 (as is done in DASH) 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.

When protocol requests show up at times when they would be inconvenient or difficult to handle, CRL queues them for later processing. In hardware-based systems, such requests are often handled by sending a negative acknowledgement (nack) back to the sender. Upon being nack-ed, requesting nodes are responsible for resending the original request. Because the overhead of receiving an active message can be significant, even in the most efficient of systems, 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. Other solutions to this problem are possible (*e.g.,* nack inconvenient requests, but use a backoff strategy when resending nack-ed messages), 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 (chosen using a simple round-robin scheme) is flushed. Doing so informs the home node that the local copy has been dropped and, if necessary, causes any changes to the data to be written back. 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 between different mappings of the same region. If a region with a valid copy of the associated data is placed in the URC 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 fully-associative, 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 [52], 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).

The `rgn_delete` function shown in Table 1 is a no-op in our current CRL implementation. We plan to implement the `rgn_delete` functionality eventually; the implementation should be straightforward, but we haven't found any pressing need to do so for the applications we have implemented to date.

Our current CRL implementation and CRL versions of several applications are available on the World Wide Web at URL `http://www.pdos.lcs.mit.edu/crl/`. In addition to the platforms described in this paper (Alewife and CM-5), this distribution can also be compiled for use with PVM [19] on a network of Sun workstations communicating with one another using TCP.

4 Experimental Platforms

This section describes the two platforms that were 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 [32] 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 7.4 Final of the CMOST operating system and version 3.3 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. All measurements were performed while the system was running in dedicated mode.

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 significantly 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 for systems like CRL, however. With interrupt-driven message delivery, the programmer is not aware of when CRL protocol messages are processed by the local node. In contrast, if polling is used, the programmer needs to be aware of when protocol messages might need to be processed and ensure that the network is polled frequently enough to allow them 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. For the communication workloads induced by the applications and problem sizes used in this study, we found that there was little or no benefit to using polling instead of interrupt-driven message delivery. We have verified this both through adding different amounts of polling (by hand) to our most communication-intensive application (Barnes-Hut) and through the use of a simple synthetic workload that allows the communication and polling frequency to be controlled independently. Thus, all CM-5 results presented in this paper were obtained by running with interrupts enabled.

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

[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.

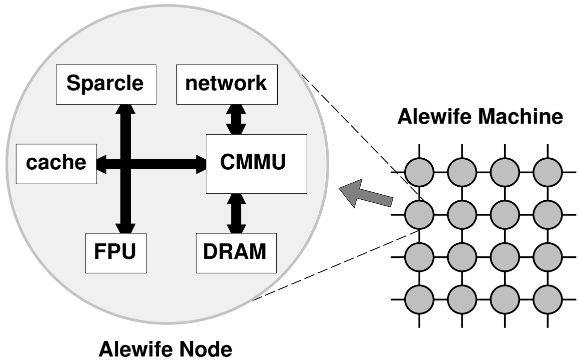

Figure 2: Basic Alewife architecture.

this problem, CRL employs a special mechanism for data transfers smaller than 256 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, 36, 47, 49]. For example, Thekkath *et al.* [48] 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 and bulk data transfer bandwidths of roughly 4.4 Mbytes/second. Since these parameters are relatively close to those for the CM-5, we expect that the performance of 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 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 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 [10]: 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. This organization is motivated by studies indicating that small-scale sharing of data is the common case [9, 38, 51]; data shared more widely is relatively uncommon. In general, Alewife's shared memory system performs quite well, enabling speedups comparable to or better than other scalable hardware-based DSM systems [1, 33].

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 [30]. 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 used on the CM-5, the round-trip time for four-word active messages on Alewife (using interrupt-driven message delivery on both ends) is approximately 14 microseconds (280 cycles). Even without correcting for the differences in clock speed, this is more than a factor of two faster than the CM-5. In our 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 causing network congestion. Currently, this transition adds approximately 12.4 microseconds (248 cycles) 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. The first subsection presents results from a simple microbenchmark that measures the latencies of various basic events. The second subsection describes the three applications (Blocked LU, Water, Barnes-Hut) that were used to evaluate the "end user" performance of CRL. The third subsection addresses the question of whether a CRL implementation built on top of high-performance communication mechanisms is capable of delivering performance competitive with that provided by hardware DSM implementations. This is accomplished by comparing the performance of Alewife CRL and Alewife's native shared memory system for the three applications. Finally, by comparing the performance of the same applications running under Alewife CRL and CM-5 CRL, the fourth subsection provides an indication

Events		CM-5 cycles	CM-5 μsec	Alewife cycles	Alewife μsec	Alewife (native) cycles	Alewife (native) μsec
Start read	hit	79	2.5	47	2.3	—	—
End read		99	3.1	51	2.6	—	—
Start read	miss, no invalidations	1925	60.2	1030	51.5	39	1.9
Start write	miss, one invalidation	3620	113.1	1760	88.0	67	3.3
Start write	miss, six invalidations	4663	145.7	3288	164.4	769	38.4

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

Events		CM-5 cycles	CM-5 μsec	Alewife cycles	Alewife μsec
Start read	miss, no invalidations	3964	123.9	1174	58.7
Start write	miss, one invalidation	5644	176.4	1914	95.7
Start write	miss, six invalidations	6647	207.7	3419	171.0

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

	Blocked LU	Water	Barnes-Hut
Source lines	1,732	2,971	3,825
`rgn_map`	27	5	31
`rgn_unmap`	30	0	29
`rgn_start_read`	19	11	17
`rgn_end_read`	19	11	15
`rgn_start_write`	11	20	22
`rgn_end_write`	11	20	27
`rgn_flush`	0	0	0

Table 4: Static count of source lines and CRL calls for the three applications.

of the sensitivity of CRL performance to changing communication costs.

Unless stated otherwise, all Alewife CRL results presented in this section include the overhead of flushing message buffers and transitioning message handlers into threads as discussed in Section 4. Because this overhead comprises 36 to 49 percent of measured Alewife CRL latencies, we expect that CRL performance (on both microbenchmarks and applications) should improve after the CMMU respin.

5.1 Basic Latencies

The following simple microbenchmark is used to measure the cost of various CRL events. 64 regions are allocated on a selected home node. Situations corresponding to desired events (*e.g.,* a start write on a remote node that requires other remote read copies to be invalidated) are constructed mechanically for some subset of the regions; the time it takes for 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 64 and then 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 140 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 830 cycles on Alewife and 2,200 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 80 cycles; the longer times correspond to 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 130 to 160 cycles larger than those for 16-byte regions; roughly three quarters of this time is due to the

	Blocked LU			Water			Barnes-Hut		
	CM-5	Alewife		CM-5	Alewife		CM-5	Alewife	
	CRL	CRL	SM	CRL	CRL	SM	CRL	CRL	SM
sequential	24.7	53.5	53.5	11.7	22.8	22.8	12.8	22.8	22.8
1 proc	25.3	54.7	53.6	13.7	24.2	22.8	24.3	34.8	23.0
32 procs	1.8	2.4	2.7	1.1	1.2	1.0	1.5	1.6	1.4

Table 5: Application running times (in seconds).

		Blocked LU		Water		Barnes-Hut	
	Events	CM-5	Alewife	CM-5	Alewife	CM-5	Alewife
CRL, 1 proc	map count (in 1000s)	84.6	84.6	—	—	983.6	983.6
	operation count (in 1000s)	84.6	84.6	269.3	269.3	992.2	992.2
CRL, 32 procs	map count (in 1000s)	2.8	2.8	—	—	30.8	30.8
	(miss rate, %)	15.3	15.3	—	—	1.1	1.1
	operation count (in 1000s)	2.8	2.8	8.7	8.7	31.3	31.2
	(miss rate, %)	14.3	14.3	7.7	9.3	4.6	4.6
	msg count (in 1000s)	1.7	1.7	2.5	3.0	5.7	5.7

Table 6: Application characteristics when running under CRL (see Section 5.2.4 for description).

overhead of flushing larger message buffers (which will be unnecessary after the CMMU respin). Even so, the fact that the differences are so small testifies to the efficiency of Alewife's block transfer mechanism.

Interestingly, these latencies indicate that with regions of a few hundred bytes in size, Alewife CRL achieves a remote data access bandwidth similar to that provided by hardware-supported shared memory. With a miss latency of 1.9 microseconds for a 16-byte cache line, Alewife's native shared memory provides a remote data access bandwidth of approximately 8.4 Mbytes/second. For regions the size of cache lines, Alewife CRL lags far behind. For 256-byte regions, however, Alewife CRL delivers 4.4 Mbytes/second (256 bytes @ 58.7 microseconds); discounting the overhead of flushing message buffers and transitioning message handlers into threads increases this to 7.9 Mbytes/second (256 bytes @ 32.4 microseconds). 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. Since the CM-5 provides less efficient mechanisms for bulk data transfer, much larger regions are required under CM-5 CRL to achieve remote data access bandwidth approaching that delivered by Alewife's hardware-supported shared memory.

5.2 Applications

While comparisons of the performance of low-level mechanisms can be revealing, end-to-end performance comparisons of real applications are far more important. Three applications (Blocked LU, Water, Barnes-Hut) were used to evaluate the performance delivered by the two different versions of CRL and compare it with that provided by Alewife's native support for shared memory. All three applications were originally written for use on hardware-based DSM systems. In each case, the CRL version was obtained by porting the original shared-memory code directly—regions were created to correspond to the existing shared data structures (*e.g.,* structures, array blocks) in the applications, and the basic control flow was left unchanged. Judicious use of conditional compilation allows a single set of sources for each application to be compiled to use either CRL (on Alewife or the CM-5) or shared memory (Alewife only) to effect interprocessor communication. Table 4 shows total source line counts (including comments and preprocessor directives) and static counts of CRL calls for the three applications.

The shared-memory versions of applications use the hardware-supported shared memory directly without any software overhead (calls to the CRL functions described in Section 3 are compiled out). For the sake of brevity, the rest of the paper uses the term "Alewife SM" to refer to this case. None of the applications employ any prefetching.

5.2.1 Blocked LU

Blocked LU implements LU factorization of a dense matrix; the version used in this study is based on one described by Rothberg *et al.* [40]. The results presented in this section are for a 500x500 matrix using 10x10 blocks.

In the CRL version of the code, a region is created for each block of the matrix to be factored; thus the size of each region—the data granularity of the application—is 800 bytes (100 double-precision floating point values). Blocked LU also exhibits a fairly large computation granularity, performing an average of approximately 11,000 cycles of useful work per CRL 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 a single processor; see Tables 5 and 6.)

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

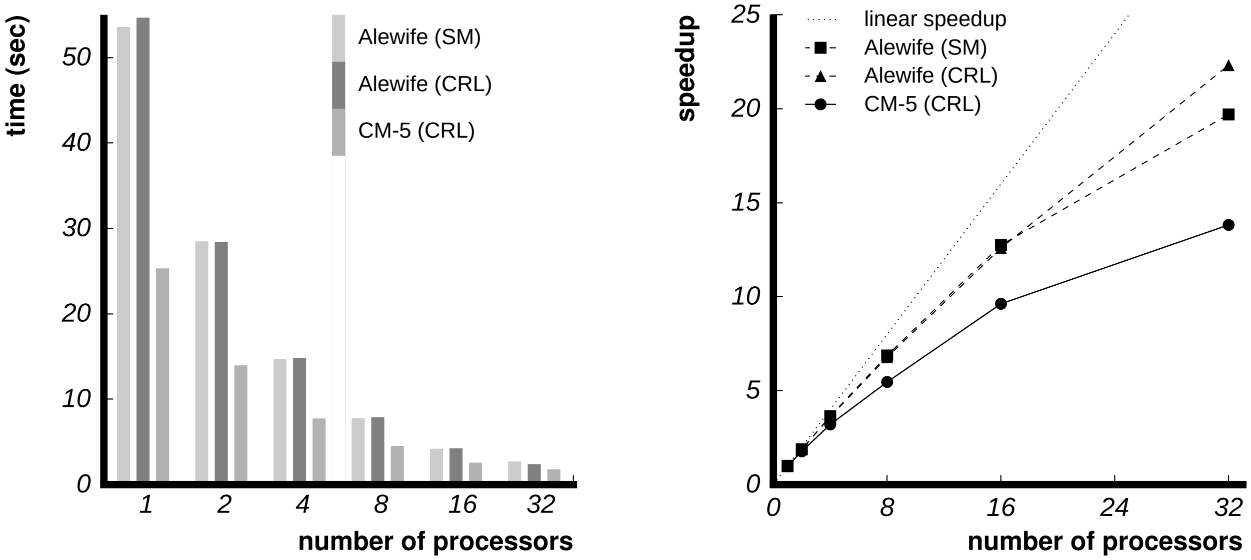

Figure 3: Absolute running time (left) and speedup (right) for Blocked LU (500x500 matrix, 10x10 blocks).

that occur during the first iteration). The results presented in this section are for a problem size of 512 molecules.

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 (via a write operation) by each processor. Although Water's data granularity is still relatively large, its computation granularity is over a factor of seven smaller than that of Blocked LU—an average of approximately 1,540 cycles per CRL operation.

5.2.3 Barnes-Hut

Barnes-Hut 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 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. The results presented in this section 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 [46].

In the CRL version of the code, a region is created for each of the octree 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 primitives provided by CRL for computing global sums, minima, and maxima.

Barnes-Hut represents a challenging communication workload. First, communication is relatively fine-grained, both in terms of data granularity (roughly 100 bytes) and computation granularity—approximately 436 cycles of useful work per CRL operation, a factor of roughly 3.5 and 25 smaller than Water and Blocked LU, respectively. Second, although Barnes-Hut exhibits a reasonable amount of temporal locality, access patterns are quite irregular due to large amounts of "pointer chasing" through the octree 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 [44, 45].

5.2.4 Performance

Table 5 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. 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 "hit" incur overhead, unlike hardware systems where hits (*e.g.,* in a hardware cache) incur no overhead.

Table 6 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 6 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); "operation count" indicates the total number of CRL operations executed (paired calls to `rgn_start_`*op* and `rgn_end_`*op*); and "msg count" shows the number of protocol messages sent and received. 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 could not be satisfied locally (without requiring interprocessor communication). All counts are average figures expressed on a per-processor basis.

Map counts 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

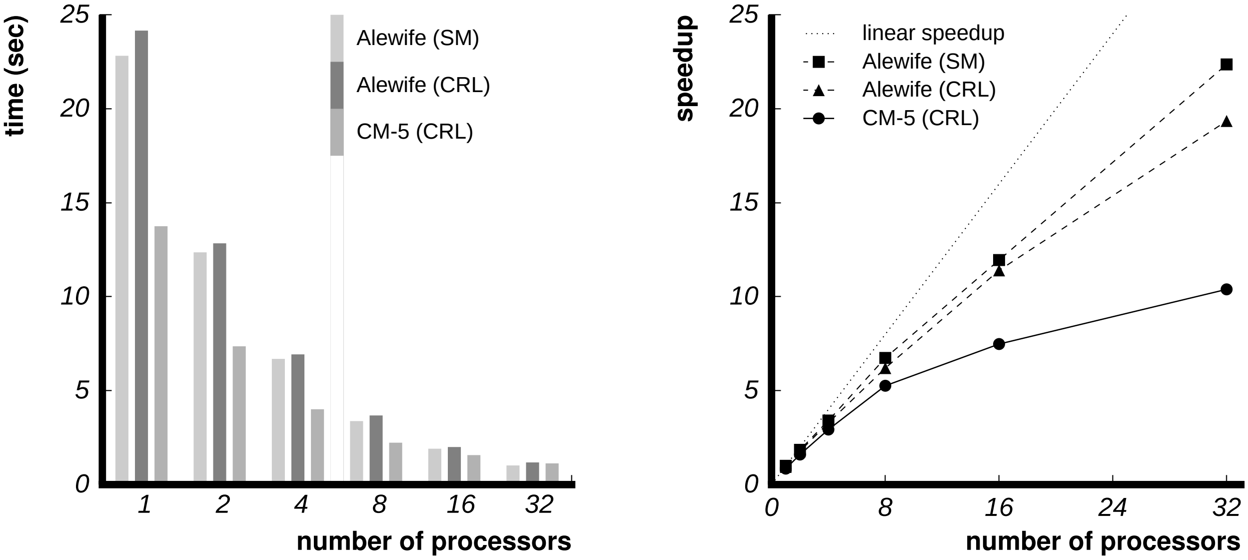

Figure 4: Absolute running time (left) and speedup (right) for Water (512 molecules).

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.

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

5.3 CRL vs. Shared Memory

In order to address the question of whether a CRL implementation built on top of high-performance communication mechanisms is capable of delivering performance competitive with that provided by hardware DSM implementations, we compare the performance of the Alewife CRL and Alewife SM versions of the three applications.

As can be seen in Figure 3, both Alewife CRL and Alewife SM perform well for Blocked LU (speedups of 22.3 and 19.7 on 32 processors, respectively). This is not particularly surprising; since Blocked LU exhibits large computation and data granularities, it does not present a particularly challenging communication workload.

Somewhat surprising, however, is the fact that Alewife CRL outperforms Alewife SM by almost 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 shared 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.

For Water, a somewhat more challenging application, both versions of the application again perform quite well; this time, Alewife SM delivers roughly 15% better performance than Alewife CRL (speedups of 22.4 and 19.3 on 32 processors, respectively).

This performance difference is primarily due to the fact that the Alewife CRL version uses three small regions to compute global sums once per iteration; each small region must "ping-pong" amongst all processors before the sum is completed. Given Alewife CRL's relatively large base communication latencies, this communication pattern can limit performance significantly as the number of processors is increased. Modifying the source code such that these global sums are computed using CRL's reduction primitives (as was already the case for Barnes-Hut) confirms this; doing so yields an Alewife CRL version of Water that delivers the same speedup at 32 processors as the Alewife SM version of the code. Because the base communication latencies for Alewife's native shared memory are significantly lower than for Alewife CRL, little or no benefit is obtained by applying the same modification to the Alewife SM version of the code (in which the same global sums were originally computed into small regions of shared memory protected by spin locks). One might expect this to change when Alewife systems with more than 32 processors become available.

For Barnes-Hut, the most challenging application used in this study, Alewife SM once again delivers the best performance—a speedup of 16.2 on 32 processors—but Alewife CRL is not far behind with a speedup of 14.5. 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 12%; experiments indicate that this gap decreases slightly for larger problem sizes).

Finally, in order to understand how different components of CRL contribute to overall running time, we developed a profiled version of the CRL library. Figure 6 shows a breakdown of running time for the Alewife CRL version of Barnes-Hut that was obtained using the profiled library for a problem size of 4,096 bodies. Normalized running time (for each bar, 1.0 corresponds to the absolute running time for Alewife CRL on that number of processors) is divided into three categories: time spent in CRL executing map/unmap code ("CRL, map"), time spent in CRL starting and ending operations ("CRL, ops"), and time spent running applica-

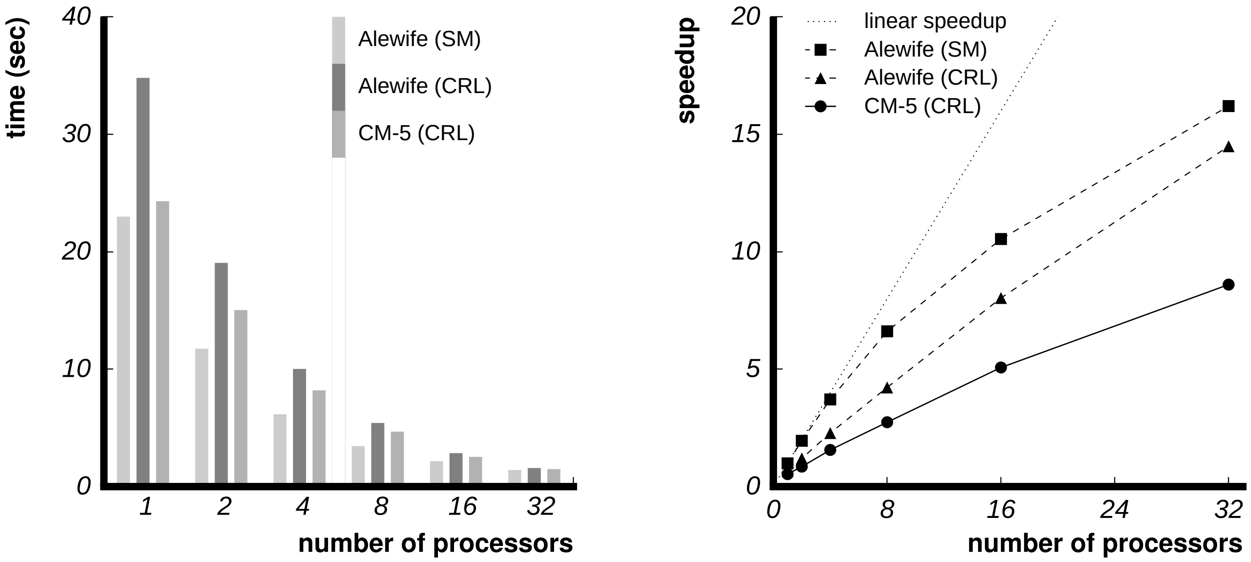

Figure 5: Absolute running time (left) and speedup (right) for Barnes-Hut (4,096 bodies).

tion code ("user"). "CRL, map" and "CRL, ops" include "spin time" spent waiting for communication events (*i.e.,* those related to calls to `rgn_map` or `rgn_start_`*op* that miss) to complete. When running on a single processor, approximately one third of the total running time is spent executing CRL code; slightly more than half of this time is spent mapping and unmapping. Not surprisingly, CRL overhead increases as number of processors is increased: at 32 processors, almost half of the total running time is spent in CRL, but now slightly less than half of the overhead is spent mapping and unmapping.

5.4 Changing Communication Costs

The results shown in the previous subsection demonstrate that when built upon high-performance communication substrates, CRL is capable of delivering performance close to that provided by hardware-supported shared memory, even for challenging applications and small problem sizes. Unfortunately, many interesting platforms for parallel and distributed computing (*e.g.,* networks of workstations) provide communication performance significantly worse than that found in Alewife.

To gauge the sensitivity of CRL's performance to increased communication costs, we compare the behavior of applications running under Alewife CRL and CM-5 CRL. Although the CM-5 is a tightly-coupled multiprocessor, current-generation network-of-workstations technology is capable of providing similar communication performance [48], so we believe that the results for CM-5 CRL are indicative of what should be possible for implementations targeting networks of workstations using current- or next-generation technology.

For Blocked LU, CM-5 CRL delivers respectable performance (a speedup of 13.8 on 32 processors), lagging roughly 30% behind the speedup achieved with Alewife CRL. Because Blocked LU uses relatively large regions, this difference can be attributed not only to the higher communication latencies on the CM-5 (1088 cycles for a simple round trip *vs.* 528 cycles for Alewife) but also to the lower bulk-data transfer performance (approximately 8 Mbytes/second *vs.* 18 Mbytes/second for Alewife).

For Water, the performance gap widens, with CM-5 CRL delivering a speedup of 10.4 on 32 processors (46% less than Alewife CRL). As was the case for Water under Alewife CRL, however, this figure can be improved upon by using reductions to compute the global sums in Water; doing so increases the speedup on 32 processors to 14.1 (37% less than the speedup on 32 processors for the same code running under Alewife CRL). The performance gap between Alewife CRL and CM-5 CRL can be attributed to the smaller computation granularity of Water (approximately 1,540 cycles of useful work per CRL operation). Even given a relatively low miss rate, 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 spite of this performance gap, CM-5 CRL performs comparably with existing mostly software DSM systems. The CM-5 CRL speedup (5.3 on eight processors) for Water (without reductions) is slightly better than that reported for TreadMarks [25], a second-generation page-based mostly software DSM system (a speedup of 4.0 on an ATM network of DECstation 5000/240 workstations, the largest configuration that results have been reported for)[2].

For Barnes-Hut, CM-5 CRL performance for Barnes-Hut lags roughly 41% behind that provided by Alewife CRL (speedups of 8.6 and 14.5 at 32 processors, respectively). As was the case with Water, this is primarily due to small computation granularity; small enough that even in the face of low map and operation miss rates (1.2% and 4.7%, respectively), the larger miss latencies of CM-5 CRL cause significant performance degradation.

As was pointed out in Section 5.2.3, the problem size used to obtain these results is one-quarter of the suggested problem size (16,384 bodies). Furthermore, even the suggested problem size is fairly modest; it is not unreasonable for production users of such codes (*e.g.,* astrophysicists) to be interested in problems with several hundred thousands bodies or more. Because larger problem sizes lead to decreased miss rates for Barnes-Hut, performance problems due to less efficient communication mechanisms on the CM-5 tend to decrease with larger problem sizes. Figure 7 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

[2]The SPLASH-2 version of Water used in this paper incorporates the "M-Water" modifications suggested by Cox *et al.* [15].

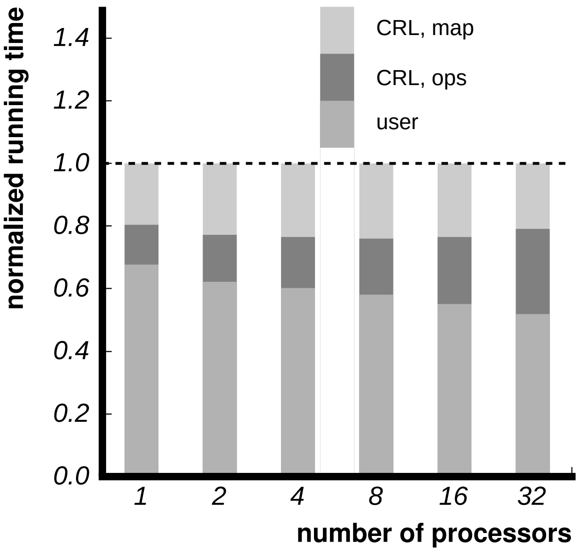

Figure 6: Breakdown of normalized running time for Alewife CRL version of Barnes-Hut (4,096 bodies).

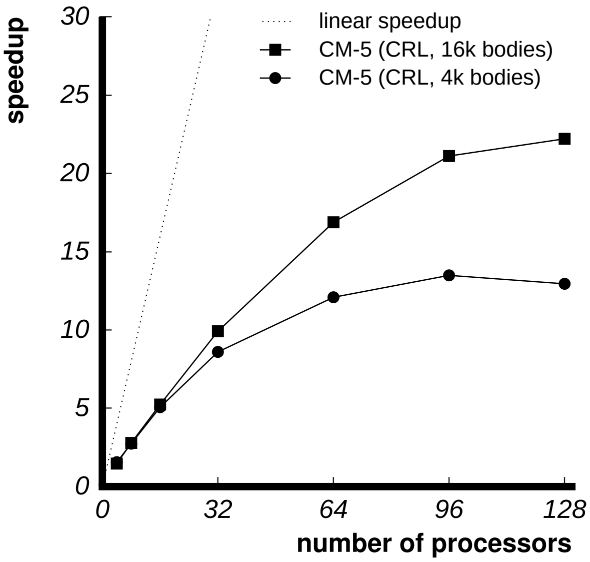

Figure 7: Barnes-Hut performance for larger problem (16,384 bodies) and machine sizes (128-node CM-5).

bodies. For the larger machine sizes (64, 96, and 128 processors), the increased problem size enables speedups 40 to 70 percent better than those for 4,096 bodies. Such results indicate that the CM-5 CRL version of Barnes-Hut may be capable of at least acceptable performance for realistic problem sizes.

5.5 Summary of Results

We draw two major conclusions from the results presented in this section. First, the CRL implementation on Alewife demonstrates that given efficient communication mechanisms, an all-software DSM system can achieve application performance competitive with hardware-supported DSM systems, even for challenging applications. For example, for Barnes-Hut with 4,096 particles, Alewife CRL delivers a speedup on 32 processors within 12% of that for Alewife SM. As discussed above, this gap should 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 be competitive with hardware systems.

6 Discussion and Future Work

Section 3.1 described three goals that guided the development of CRL; we believe 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 criterion: porting CRL to other message passing environments (*e.g.,* workstations communicating with one another using TCP) has proven to be straightforward. Finally, the performance results shown in 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 requires some additional programmer effort, but it 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 (see Table 4). 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 [18]. 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 would likely have broad impact across the spectrum of distributed systems.

7 Related Work

This section provides a brief overview of other research related to CRL; it is divided into three subsections. The first subsection discusses region- and object-based software DSM systems. The second subsection describes other software DSM work, including page-based systems. Finally, the third subsection discusses other research that provides comparative results (*e.g.,* all-software *vs.* mostly software, message passing *vs.* shared memory).

7.1 Region- and Object-Based Systems

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 [41]; the same basic notion of synchronized access ("operations") to regions ("objects") also exists in other programming systems for hardware-based DSM systems (*e.g.,* COOL [12]). The Shared Regions work arrived at this interface from a different set of constraints, however: 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 similar implementation techniques and defining `rgn_map` and `rgn_unmap` to be null macros.

Chandra *et al.* [11] propose a hybrid DSM protocol in which region-like annotations are used to demark access to regions of shared data. Coherence for regions thus annotated 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.

Of all other software DSM systems, Cid [37] is perhaps closest in spirit to CRL. Like CRL, Cid 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 used on a small cluster of workstations connected by FDDI[3]. CRL runs on two large-scale platforms and has been shown to deliver performance competitive with hardware DSM systems.

Several all-software DSM systems that employ an object-based approach have been developed (*e.g.,* Amber [14], Orca [3], Concert [24]). Like these systems, CRL effects 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, 24] or only allow the use of a subset of an existing one [14]. In contrast, CRL is not language specific; the basic CRL interface could easily be provided in any imperative programming language.

Scales and Lam [42] 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. Like CRL, SAM is implemented as a portable C library. Informal discussion with Scales indicates that SAM delivers approximately 35% better speedup than CRL for Barnes-Hut when running on the CM-5 with a problem size of 16,384 bodies. This advantage is enabled by additional information about communication patterns provided through SAM's new communication primitives. SAM's performance edge comes at a cost, however: Because the primitives SAM offers are significantly different than "standard" shared memory models, converting existing shared-memory applications to use SAM is likely to be more difficult than converting them to use CRL.

7.2 Other Software DSM Systems

TreadMarks [25] is a second-generation page-based (mostly software) DSM system that implements a release consistent memory model. Unlike many page-based systems, TreadMarks is implemented entirely in user space; virtual memory protection mechanisms are manipulated through library wrappers around system calls into the kernel. Since these virtual memory mechanisms and associated operating system interfaces are relatively standard in current commodity workstations, TreadMarks is fairly portable. There are interesting platforms (*e.g.,* CM-5, Alewife) that lack the support required to implement TreadMarks, however; we believe that this will continue to be the case. In addition, the software overhead of systems like this (*e.g.,* from manipulating virtual memory mechanisms and computing diffs) can be large enough to significantly impact delivered application performance [17].

Midway is a software DSM system based on entry consistency [5]. As discussed in Section 3.3, CRL's programming model is similar to that provided by Midway. An important difference, however, is that Midway requires a compiler that can cull user-provided annotations that relate data and synchronization objects from the source code and provide these to the Midway run-time system. By bundling an implicit synchronization object with every region, CRL obviates the need for special compiler support of this sort. Both mostly software and all-software versions of Midway have been implemented [54]. To the best of our knowledge, Midway has only been implemented on a small cluster of workstations connected with an ATM network.

A number of other approaches to providing coherence in software on top of non-cache-coherent shared-memory hardware have also been explored [16, 28]. 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).

7.3 Comparative Studies

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 [10, 21, 53], mostly software *vs.* all-software [43, 54]), but to our knowledge, only Cox *et al.* [15] 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).

[3] Rishiyur S. Nikhil, personal communication, March 1995.

Klaiber and Levy [27] 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, 13] or looked at the performance gains made possible by augmenting a hardware DSM system with message passing primitives [29]. Such research addresses a different set of issues than those discussed in this paper, which takes a distributed shared memory programming model as a given and provides a controlled comparison of hardware and software implementations.

Finally, Schoinas *et al.* [43] 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. To achieve this, CRL requires the programmer to define regions and delimit accesses to them using annotations.

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 on the CM-5, with communication performance on par with that available using current-generation network of workstations technology, CRL performs well. Furthermore, we show that given high-performance communication mechanisms, CRL is capable of delivering performance competitive with that provided by hardware-supported DSMs, even for challenging applications and small problem sizes. For Barnes-Hut, Alewife CRL delivers performance on 32 processors within 12% of that provided by Alewife's hardware-supported shared memory.

We therefore conclude that for CRL to be competitive with hardware-supported DSMs, high-performance communication mechanisms are necessary: the communication performance of the CM-5 and current-generation network of workstation technology is probably not sufficient. This general conclusion is not specific to CRL; we expect that it applies to other software DSM approaches as well.

The results presented in this paper suggest that CRL or other all-software approaches can be used to provide portable and efficient distributed shared memory programming environments without any need for hardware support beyond that required for high-performance message-based communication.

9 Acknowledgements

Anant Agarwal, David Chaiken, John Guttag, Wilson Hsieh, David Kranz, Kevin Lew, Dan Scales, and Willy Zwaenepoel provided helpful feedback and commentary on earlier versions of this work; comments from the program committee and our paper's shepherd were also quite useful. John Kubiatowicz's tireless efforts to provide a better software interface to Alewife's message passing hardware proved invaluable. Joseph Adler and Sandeep Gupta developed the TCP/UNIX port of CRL.

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*, pages 2–13, June 1995.

[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 and Anne Rogers. Software Caching and Computation Migration in Olden. In *Proceedings of the Fifth Symposium on Principles and Practices of Parallel Programming*, pages 29–38, July 1995.

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

[9] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, pages 41–58, June 1990.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, May 1993.

[18] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[19] 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.

[20] 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*, pages 15–26, June 1990.

[21] 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.

[22] Wilson C. Hsieh. *Dynamic Computation Migration in Distributed Shared Memory Systems*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1995.

[23] 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.

[24] Vijay Karamcheti and Andrew Chien. Concert – Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Proceedings of Supercomputing '93*, pages 598–607, November 1993.

[25] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[26] Kendall Square Research. KSR-1 Technical Summary, 1992.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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*, pages 272–285, June 1992.

[33] 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.

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

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

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

[37] 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.

[38] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 138–147, June 1990.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Jaswinder Pal Singh, Anoop Gupta, and John L. Hennessy. Implications of Hierarchical N-Body Techniques for Multiprocessor Architecture. *ACM Transactions on Computer Systems*, pages 141–202, May 1995.

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

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

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

[48] 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.

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

[50] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[51] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, April 1989.

[52] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[53] 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.

[54] 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.