Performance Implications of Communication Mechanisms in All-Software Global Address Space Systems

Beng-Hong Lim[†], Chi-Chao Chang[⋆], Grzegorz Czajkowski[⋆] and Thorsten von Eicken[⋆]

[†]T.J. Watson Research Center
IBM Corporation
Yorktown Heights, NY 10598

[⋆]Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Global addressing of shared data simplifies parallel programming and complements message passing models commonly found in distributed memory machines. A number of programming systems have been designed that synthesize global addressing purely in software on such machines. These systems provide a number of communication mechanisms to mitigate the effect of high communication latencies and overheads. This study compares the mechanisms in two representative all-software systems: CRL and Split-C. CRL uses region-based caching while Split-C uses split-phase and push-based data transfers for optimizing communication performance. Both systems take advantage of bulk data transfers.

By implementing a set of parallel applications in both CRL and Split-C, and running them on the IBM SP2, Meiko CS-2 and two simulated architectures, we find that split-phase and push-based bulk data transfers are essential for good performance. Region-based caching benefits applications with irregular structure and with sufficient temporal locality, especially under high communication latencies. However, caching also hurts performance when there is insufficient data reuse or when the size of caching granularity is mismatched with the communication granularity. We find the programming complexity of the communication mechanisms in both languages to be comparable. Based on our results, we recommend that an ideal system intended to support diverse applications on parallel platforms should incorporate the communication mechanisms in CRL and Split-C.

1 Introduction

Shared-memory provides a simple, intuitive model for parallel programming and is widely used on multiprocessors with hardware support for global memory addressing. However, as networked clusters of workstations and PCs become commonplace, there is a tremendous incentive to harness their collective computation power to run parallel programs. On such systems, message passing comprises the default programming model and there is typically no hardware support for shared memory. To complement message passing and ease programming, a number of alternative programming systems synthesize global addressing in software using the underlying hardware messaging primitives.

Such all-software global address space systems have to contend with high communication latencies and overheads in a clustered environment, where overhead is usually in the order of a few microseconds, and latency is in the order of tens of microseconds.[1] Clearly, a naive implementation of global addressing that fetches a single word of data from a remote node at each data access will not perform well. Thus, a number of mechanisms have been developed to tolerate high communication latencies and overheads in these systems.

This paper investigates the performance implications of the major mechanisms in all-software systems to tolerate latencies and overheads: caching, bulk communication, split-phase communication, and push-based (sender-initiated) communication. We use CRL [10] and Split-C [7], two representative systems that provide these mechanisms. CRL caches program-designated memory regions to exploit temporal and spatial locality. Bulk communication occurs implicitly when using large memory regions. Split-C provides routines for bulk communication, split-phase communication and push-based communication.

We compare the performance of a set of applications written in both languages on an IBM SP2 [1], a Meiko CS-2 [9], and on two simulated machine architectures. We created the application set by taking applications that were originally written in CRL or Split-C and rewriting them to use communication mechanisms provided by the other language. This prevents a bias towards either language, and ensures that the computation portion of the applications are identical. We also make further modifications using the mechanisms available within each language to isolate the effects of individual communication primitives.

This paper makes the following contributions:

- it investigates mechanisms for optimizing performance in all-software globally-addressed programming systems, such as bulk transfers, caching, and split-phase communication,

This work has been sponsored by IBM under the joint project agreement 11-2691-A and University Agreement MHVU5851, and by NSF under contracts CDA-9024600 and ASC-8902827. Chi-Chao Chang is supported in part by a doctoral fellowship (200812/94-7) from CNPq/Brazil.

To appear in the Sixth ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '97), Las Vegas, Nevada, June 1997.

[1] *Overhead* refers to the cycles spent on compute processors to send and receive messages, and *latency* refers to the delay between sending a message and receiving it on the remote end.

- it compares the performance impact of the different mechanisms on a set of non-trivial applications,
- it details our experience with using globally-addressed programming systems to implement and optimize parallel applications,
- it demonstrates the benefits and limitations of region/object-based caching and of bulk data copying, and motivates the need to incorporate both features into a single programming environment.

The rest of this paper is organized as follows. Section 2 describes all-software global address space programming systems and compares CRL and Split-C. Section 3 describes our experimental settings. Section 4 presents and analyzes the results of our experiments and the utility of each communication mechanism for enhancing performance. Section 5 discusses related work. Finally, Section 6 concludes the paper.

2 Overview of CRL and Split-C

Programming systems that provide global addressing without relying on hardware support may be classified into two groups: all-software distributed shared memory (DSM) systems that provide cache-coherent access to globally shared objects/data (*e.g.*, CRL [10], Midway [21], Orca [2] and SAM [16]) and systems that provide primitives to transfer shared data using global pointers and arrays (*e.g.*, Split-C [7], Global Arrays [14], CC++ [4] and High Performance Fortran [8]).

These all-software global address space programming systems provide mechanisms to tolerate high communication latencies and overheads present in off-the-shelf hardware: caching, bulk communication, split-phase communication and push-based communication. Caching replicates data coherently in order to take advantage of temporal and spatial locality. Bulk communication amortizes the fixed cost of communication by transferring large amounts of data in a single message. Split-phase communication decouples initiation and completion of communication, and allows computation to overlap communication. Finally, push-based, or sender-initiated communication allows a producer to send data to consumers as soon as it is ready and potentially before it is needed.

To study the effectiveness of these mechanisms, our investigation uses CRL and Split-C, two representative programming systems from each of the two groups. Since CRL and Split-C are both extensions of C, compiled with a common compiler (`gcc`), and depend on a common communication layer (Active Messages [19]), performance differences between the two can be isolated to the different communication mechanisms used by each system.

2.1 CRL

CRL is an all-software DSM system that relies on the programmer to identify regions as logical units of caching (for example, one may designate a matrix column as a region). Regions are created by calling `region_create(size)`, which returns a globally unique region identifier. Before accessing a region, `region_map(region_id)` must be called to map the data into the local address space and to obtain a local pointer to the data. To preserve coherence, reads to data within a region must be bracketed by `region_start_read` and `region_end_read` which implicitly acquire and release a read lock for the region. Similarly, modifications to data must be bracketed by `region_start_write` and `region_end_write`.

CRL is representative of all-software DSM systems that provide global naming and coherent caching of application-defined objects or regions. Compared with other systems, CRL does not rely on compiler or operating system assistance or on a new language definition. It is implemented as a portable user-level library that relies only on the ability to send and receive messages. CRL's minimalist approach allows us to focus on the caching mechanisms provided by all-software DSM systems.

2.2 Split-C

Split-C is a parallel extension to C, with a small set of operators to allow efficient programming of SPMD programs. Processes communicate by accessing global data, either by dereferencing global pointers for scalar types, or by using bulk transfer to copy contiguous blocks of global data. To facilitate data partitioning, Split-C provides spread arrays which distribute data block cyclically across processors.

Access to global data may be blocking or split-phase. Completion of split-phase data transfers is determined by explicit `sync()` operations that block the caller until all data transfers complete. Stores to remote data support efficient one-way communication and remote event notification. The language also provides means of performing simple remote actions atomically.

Split-C is representative of systems that provide remote access to global data. By completely leaving the burden of orchestrating remote data access to the programmer, it provides minimal support for global addressing and allows us to focus on its mechanisms for accessing and transferring remote data.

2.3 Comparison

Table 1 summarizes the communication mechanisms supported by Split-C and CRL.

Communication mechanism	*CRL*	*Split-C*
caching	yes	no
bulk communication	yes	yes
split-phase communication	no	yes
push-based communication	no	yes

Table 1: Communication mechanisms in CRL and Split-C.

CRL, to a large extent, provides a convenient abstraction of shared memory. However, the programmer needs to identify all data areas that will be shared. Even a small piece of data, like a global integer counter, needs to be identified as a region and manipulated as such. Although this may force a programmer to think carefully about data sharing, we found it to be rather time consuming. Split-C, on the other hand, allows access to any global data by simply dereferencing pointers.

CRL eliminates many unnecessary data transfers with its built-in cache coherence protocols. Split-C does not provide caching, although the programmer may still exploit temporal locality by explicitly determining when previously copied data may be safely reused. The difficulty of performing such explicit caching in Split-C varies with the application.

Split-C allows precise control of data movement by providing push-based, one-way stores and split-phase operations. These mechanisms allow a programmer to optimize communication patterns, and result in better performance for some applications when compared to CRL which forces all communication to occur through coherent caching of region data. Although the design focus of CRL is to provide a cache-coherent DSM system, explicit communication primitives could be incorporated in future versions of CRL.

Both CRL and Split-C provide bulk communication. In Split-C it occurs via explicit communication library calls, while in CRL it occurs when large regions are cached. In CRL, a region's size is specified and fixed at creation time. In cases where the logical unit of sharing varies dynamically, the fixed size can either lead to unnecessary data transfers or to false sharing. On the other hand, once a region is created, the programmer can manipulate it without thinking about its size. In contrast, Split-C requires the programmer to specify the size of the remote data for every bulk transfer. Of course, one can also provide an abstraction layer on top of Split-C that associates sizes with source and destination buffers.

3 Experiments

In order to compare the efficiency of the communication mechanisms in CRL and Split-C, we run a total of five applications, some of them in multiple versions, on two existing multicomputers and two simulated machines. This section describes each of the platforms and their Active Message layers, as well as the applications.

3.1 Hardware Platforms

Meiko CS-2 The Meiko CS-2 [9] is a multicomputer where each node contains a 40 MHz three-way superscalar SuperSparc processor and a custom network adapter. The network adapter contains a special-purpose "Elan" network processor that is integrated with the network interface and DMA controller. The network adapter is attached to the memory bus and appears as a memory-mapped device. The compute processor can issue commands to and receive responses from the network processor via user-level memory instructions. The network processor has only modest processing power and no general purpose cache, so instructions and data are accessed from main memory. The custom network is comprised of two 4-ary fat-trees.

Meiko's network processor provides a user-level interface to directly read from and write to the address space of a process on a remote node. Meiko Active Messages bypass this interface and uses the network processor directly to optimize performance [17]. It achieves a peak bandwidth of 39 MB/s and an `am_request/reply` round-trip time of 11 μs.

IBM SP2 The IBM SP2 [1] is a multicomputer comprised of RS/6000 workstation-class nodes connected by a custom network. Each node has a 66MHz POWER2 processor and a custom network adapter with a built-in coprocessor and DMA controller. The network adapter is attached to a 32-bit MicroChannel I/O bus, and interfaces to a custom multistage interconnection network.

The SP2 network adapter provides a user-level interface to a pair of send and receive FIFO queues synthesized by microcode running on the coprocessor on the network adapter. The Active Message layer designed at Cornell University for the SP2 [5] interfaces directly to these queues and achieves a peak bandwidth of 34 MB/s and an `am_request/reply` round-trip time of 51 μs.

RMC1 and RMC2 RMC1 and RMC2 simulate an architecture that supports remote memory access (PUT/GET) and remote queue operations directly in hardware [12]. Remote memory access allows a process to read and write memory in the address space of another, possibly remote, process. Remote queues allow a process to enqueue and dequeue data in the address space of another process. Both RMC1 and RMC2 use processing nodes similar to those of the SP2. RMC1 and RMC2 differ only in the network latency, which is an aggressive 0.1μs on RMC1 and a slow 100 μs on RMC2. These simulated architectures allow us to investigate the effect of different communication architectures and of different communication speeds.

We implemented an Active Message layer on RMC1 and RMC2 using the remote memory access and remote queue hardware primitives. `am_request` and `am_reply` messages are queued on remote nodes using the remote queue primitives. `am_get` uses a GET operation to copy remote data locally. `am_store` uses a PUT operation to copy local data to a remote destination and remote queue primitives to enqueue a completion handler on the destination node. The Active Message layer on RMC1 achieves a 17μs round-trip delay for `am_request/reply`, and a peak bandwidth of 500 MB/s. On RMC2, it achieves a 217μs round trip delay and peak bandwidth of 500 MB/s.

Table 2 summarizes the features of the machines. AM half-power point refers to the minimum active message size required to achieve more than half the peak bandwidth.

3.2 Benchmark Applications

We use five parallel benchmarks to compare the performance of Split-C and CRL and to evaluate their communication mechanisms: matrix multiply (*mm*), Fast Fourier Transform (*fft*), blocked LU decomposition (*lu*), water molecule simulation (*water*) and a Barnes-Hut N-body simulation (*barnes*). We use the code for *lu*, *water* and *barnes* from the CRL distribution and the code for *fft* and *mm* from examples in the Split-C distribution. We convert the CRL code to use the communication mechanisms in Split-C and vice versa. This results in a base set of benchmarks described below. We also modify some of the benchmarks to better understand and quantify the performance impact of particular features of the systems under study.

Table 3 lists the benchmarks and their input parameters for the experiments.

Matrix Multiply This is the simplest of all the benchmarks. It multiplies two matrices, A and B, which are distributed in a block-

Machine	*CPU type*	*AM Round-trip Latency*	*AM Peak Bandwidth*	*AM Half-power point*
IBM SP2	66 MHz POWER2	51μs	34MB/s	2.8 KB
Meiko CS-2	40 MHz Sparc-20	25μs	39MB/s	2 KB
RMC1	66 MHZ POWER2	17μs	500MB/s	8 KB
RMC2	66 MHz POWER2	217μs	500MB/s	64 KB

Table 2: Comparison of performance characteristics of the IBM SP2, the Meiko CS-2 and the simulated RMC1 and RMC2 machines.

Benchmark	*First set of runs*	*Second set of runs*
mm	512x512 matrix, 16x16 block	512x512 matrix, 128x128 block
fft	1 million points	2 million points
lu	512x512 matrix, 4x4 block	512x512 matrix, 16x16 block
water	64 molecules	512 molecules
barnes	512 bodies	-

Table 3: Benchmark parameters.

cyclic fashion. The result C shares no memory locations with A or B. Needed blocks are fetched pairwise, just before they are used for multiplication. Since every processor fetches the same blocks of A and B repeatedly, *mm* exposes an opportunity for caching. The CRL version (*mm/crl*)[2] opens appropriate regions for reading matrix blocks. The Split-C version, (*mm/sc*) issues two non-blocking bulk get requests for each matrix block, followed by a call to `sync()` to ensure completion of the bulk transfer requests.

Fast Fourier Transform This benchmark computes the n-input butterfly algorithm for the discrete one-dimensional FFT problem using P processors. The algorithm is divided into three phases: (i) $log(n) - log(P)$ local FFT computation steps using a cyclic layout where the first row of the butterfly is assigned to processor 1, the second to processor 2, and so on; (ii) a data remapping phase towards a blocked layout where the n/P rows are placed on the first processor, the next n/P rows on the second processor, and so on; and (iii) $log(P)$ local FFT computation steps using the blocked layout. In the first and third phases, each processor is responsible for transforming n/P elements.

The base Split-C version (*fft/split-c*) uses a spread vector to represent the input elements. In effect, each processor allocates a single n/P-element vector to represent its portion of the butterfly. Communication occurs only in the data remapping phase where each processor uses bulk communication to send a n/P^2-element chunk of data to each remote processor. The communication is staggered to avoid hot spots at the destination.

The base CRL version of FFT (*fft/crl*) is based on *fft/split-c*. The primary modification consists of using a vector of P regions, where each region contains n/P^2 elements, to represent the n/P-element vector on each processor. The region size of n/P^2 elements is chosen to match the required data transfer size and minimize communication bandwidth during the data remapping phase. The price for such a layout is that the n/P-element vector is no longer allocated contiguously in memory and extra index calculations are required during the local computation phases.

[2]We use the notation *application/system* to refer to the implementation of an application on a particular system.

Blocked LU decomposition This application implements *in-situ* factorization of a dense matrix as described in [15]. The communication and computation structure of this application is as follows: The matrix is divided up into blocks distributed among processors. Every step comprises three substeps, between which processors synchronize with a barrier. First, the pivot block (I, I) is factored by its owner-processor. Second, all processors which have blocks in the I-th row or I-th column obtain the updated pivot block. Third, all internal blocks are updated. An important observation about the benchmark is that all remote blocks requested in a given substep need to be fetched, since they were modified in preceding substeps.

The base CRL version *lu/crl* uses an array of regions to represent the matrix to be factored, where each region represents a single block of the matrix. Matrix blocks are transferred between processors as part of the cache coherence protocol when they are read and written. The base Split-C version (*lu/sc*) uses one-way stores for explicitly transferring pivot blocks (a feature not available in CRL) and prefetches all blocks before beginning the third substep. No prefetching occurs in *lu/crl*.

Water Water is an N-body molecular dynamics application that computes the forces and energies of a system of water molecules. The computation iterates over a number of steps, and every step includes computing the intra- and inter-molecular forces for molecules contained in a "cubical" box, which runs in $O(n^2)$ time. A predictor-corrector method is used to integrate the motion of the water molecules over time. The total potential energy is calculated as the sum of intra- and inter-molecular potentials. The main data structure is an array of molecules which is distributed statically across all processors. The intra-molecule interactions are computed locally, whereas the inter-molecule ones require reads and writes of remote data.

In the base CRL version (*water/crl*), each molecule is represented as a 672-byte region, and read and write operations on the molecules are bracketed by the appropriate calls to read and write the region. The Split-C version (*water/sc*) issues atomic reads and writes to access and update the remote molecules.

Barnes-Hut This application simulates the evolution of a system of bodies under the influence of gravitational forces using the hier-

archical N-body algorithm proposed by Barnes and Hut [18]. The computation is highly irregular and the communication is relatively fine-grained: a distributed oct-tree is built up with the bodies at the leaves. Each tree node is less than 150 bytes. The algorithm traverses many pointer chains making the remote access pattern quite irregular. In addition, the algorithm is synchronization intensive: during the tree-building phase, tree nodes are locked when inserting a new leaf node that represents a body. During the simulation phase, each body is locked when its parameters are updated.

The CRL version of Barnes-Hut (*barnes/crl*) creates a region for every tree node. An element is locked by opening the corresponding region for writing. The base Split-C version (*barnes/sc*) simply follows the structure of the CRL version and has identical reference patterns. *barnes/sc* uses atomic integer writes and reads to remote locations for locking and unlocking. Although we could have attempted to aggregate multiple data transfers into a single transfer to coarsen the communication granularity in *barnes/sc*, the irregular communication pattern would require a significant programming effort.

3.3 Modifications to base benchmarks

We further modify several of the benchmarks to gain better insight into the performance implications of the features of each system, and to quantify the cost of different communication mechanisms.

Both *mm/sc* and *barnes/sc* have large amounts of data that is repeatedly read from remote locations even though the local copy is not stale. Two new versions, *mm-cache/sc* and *barnes-cache/sc* perform explicit caching in Split-C to evaluate the potential benefits.

Experimenting with *lu* led to three new versions. The first, *lu-pull/sc*, uses bulk gets instead of one-way stores to determine the cost of requesting the data. The second, *lu-c/sc*, requests blocks just before they are needed instead of prefetching all internal blocks before the computation begins. *lu-c/sc* has a communication structure identical to *lu/crl*. Finally, the impact of prefetching on CRL is evaluated with *lu-pref/crl* which prefetches internal blocks. *lu-pref/crl* and *lu-pull/sc* have similar communication patterns.

Finally, we optimize the read-phase in *water/sc* by replacing the atomic read requests with selective prefetching, where selected data of remote molecules are bundled and fetched from their respective processors prior to local computing. The resulting version is *water-slpf/sc*.

4 Results

This section analyzes the performance of the base benchmarks and their modified versions on an IBM SP2, a Meiko CS-2 and on simulations of the RMC1 and RMC2 architectures.

Figures 1 through 4 present the execution times of the base benchmark set on 8 processors of each of the four multiprocessor platforms, normalized to the Split-C versions. The execution time is split into CPU time, synchronization time (e.g., barriers and explicit *sync()* statements) and the time taken to transmit data. For CRL, the time spent for the cache coherence protocol is also measured by instrumenting the region read and write functions.[3] All timings are obtained using real-time clocks accessible in fewer than 20 machine instructions. Time spent in data transfer was measured in the Active Message layer and the synchronization primitives were instrumented in both the Split-C and the CRL libraries. Finally, the CPU time is the difference between the total execution time and the sum of the other components. It is important to note that the same compiler (gcc) is used for both CRL and Split-C and that the compute kernels in the corresponding benchmarks are the same.

The results show that the applications fall into two groups: i) those that benefit from CRL's caching (*barnes, mm*), and ii) those that perform better with Split-C's explicit data transfers (*fft, lu* and *water*).

Barnes-Hut *barnes/crl* runs 1.7 times faster than *barnes/sc* on the SP2, and 2.2 times faster on RMC2. This happens mainly because *barnes/sc* transfers about 10 times more data than *barnes/crl*. This is primarily due to the lack of caching in Split-C. Another reason for this difference is padding of data in Split-C for programming convenience. About 9% of the data sent in *barnes/sc* is due to padding: the elements of the oct-tree are of different types and sizes, and the Split-C version pads them all to a common maximum size in order to avoid a type check each time an element is requested. CRL avoids this problem because the region size is permanently bound to the region itself.

However, if remote access latency is low enough, we find that the performance advantage of caching diminishes and *barnes/sc* is only 13% slower than *barnes/crl* on the Meiko CS-2, and actually runs 1.5 times faster than *barnes/crl* on RMC1. This result emphasizes the point that remote access latency has to be high enough to justify the overhead of caching data.

In parts of *barnes/sc*, remote structures are repeatedly read across the network, even though they may not have been modified. To avoid repeated network accesses, *barnes-cache/sc* caches selected remote reads that occur frequently. This improves performance significantly: *barnes-cache/sc* is only about 1.1 times slower than *barnes/crl* on the SP2. However, determining which objects to cache and when to cache was not an easy programming task. Certainly, automatic support for caching in Split-C would be very useful for this application.

Matrix Multiply In *mm*, the frequent reuse of remote matrix blocks that are cached locally allows *mm/crl* to run about 1.1 times faster than *mm/sc* that uses bulk data transfer to copy remote matrix blocks repeatedly. The exceptions are small block sizes (16) on RMC2 and both block sizes (16 and 128) on the Meiko, where CRL performs slightly worse. In case of RMC2 this is explained by the very high communication latency which makes the cost of the coherence protocol non-negligible.

The caching version of matrix multiplication in Split-C (*mm-cache/sc*) performs as well as *mm/crl*. Since the communication pattern in matrix multiplication is very regular, determining which blocks to cache is straightforward.

FFT In *fft*, remote blocks of data are read only once during the remapping phase and there is no temporal locality that can be exploited with caching. In fact, the region-based caching in CRL

[3] Due to instrumentation problems, we were unable to obtain the coherence cost on the Meiko CS-2. Instead, the coherence cost is included in the CPU time.

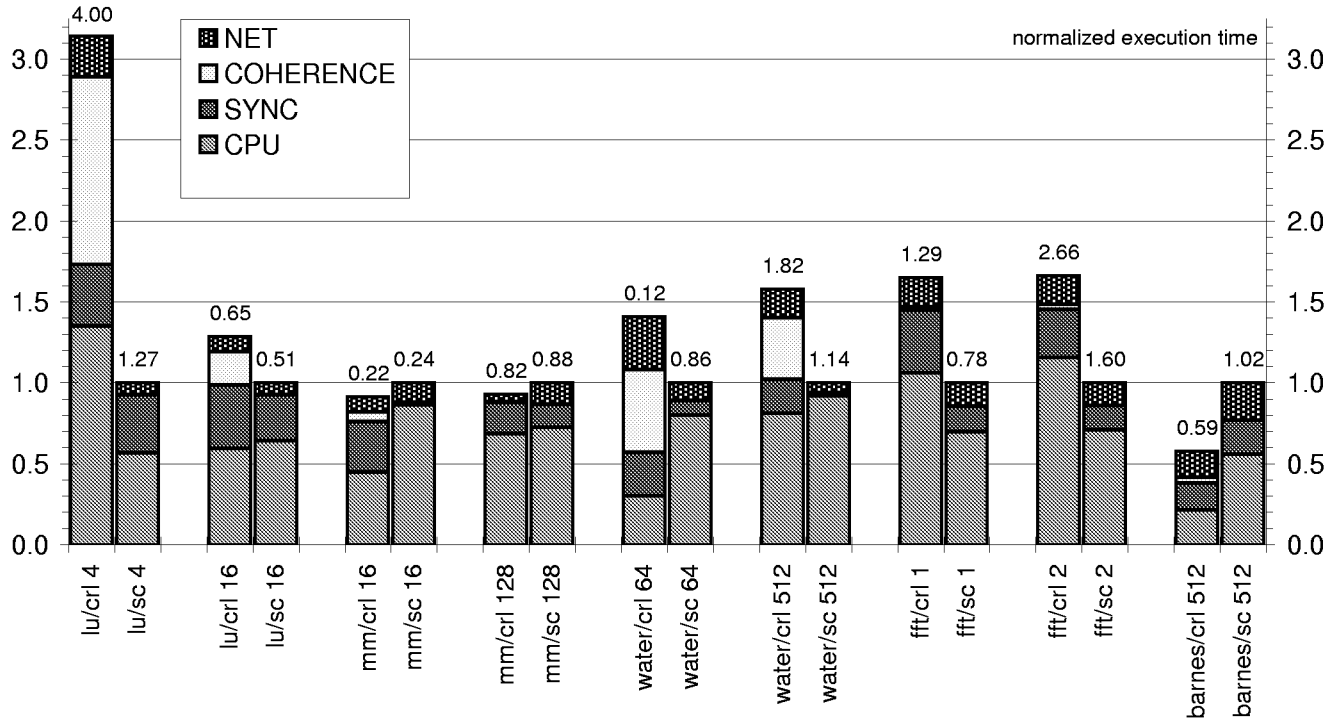

Figure 1: Relative execution times of the base benchmarks on an 8-node SP2, normalized to Split-C. The absolute execution time in seconds is shown above each bar.

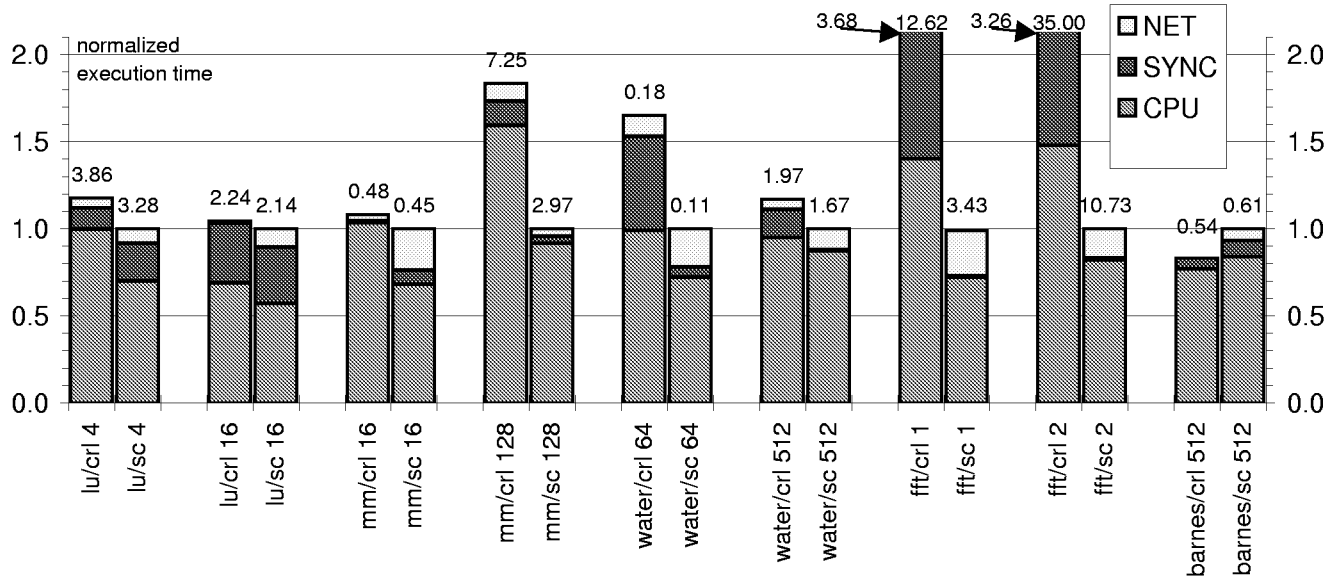

Figure 2: Relative execution times of the base benchmarks on an 8-node Meiko CS-2,normalized to Split-C. The absolute execution time in seconds is shown above each bar.

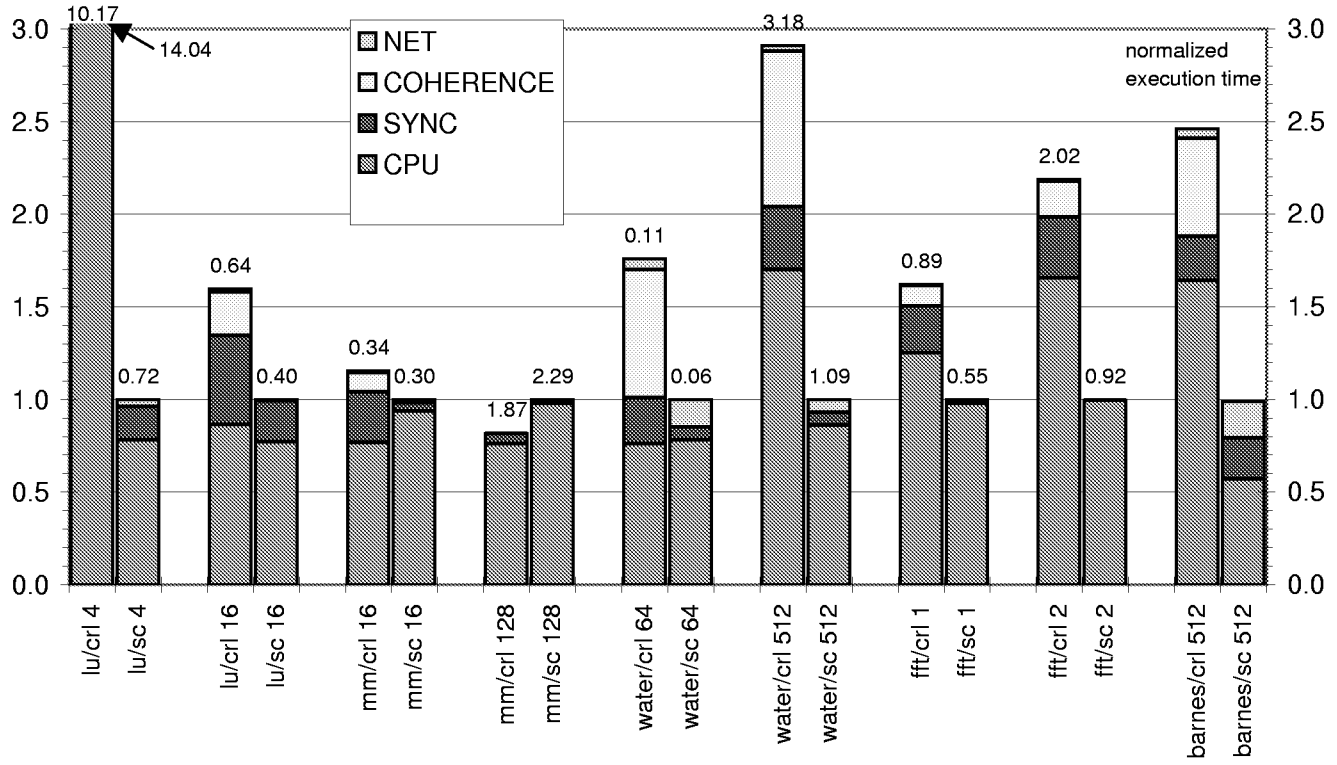

Figure 3: Relative execution times of the base benchmarks on an 8-node RMC1, normalized to Split-C. The absolute execution time in seconds is shown above each bar.

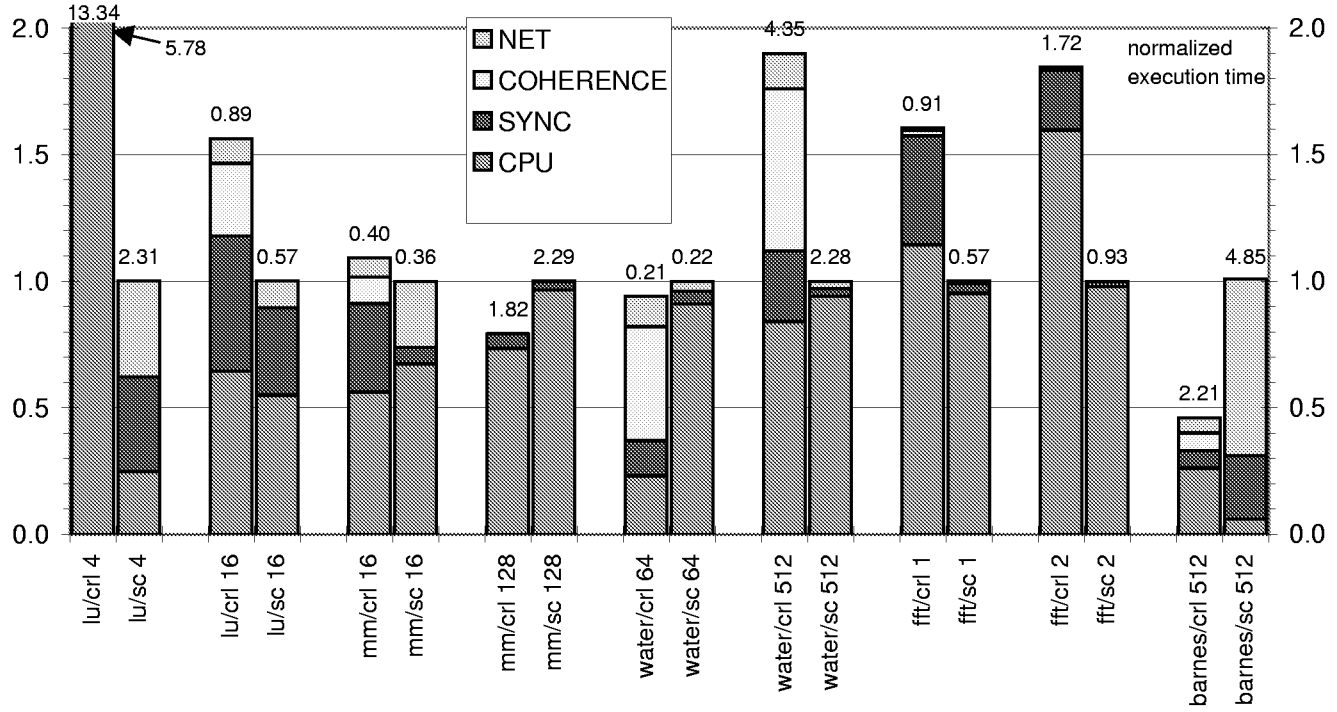

Figure 4: Relative execution times of the base benchmarks on an 8-node RMC2, normalized to Split-C. The absolute execution time in seconds is shown above each bar.

hurts performance significantly due to the extra overhead of index calculations during the local computation phase: additional measurements show that *fft/crl* takes about 1.5 times longer than *fft/sc* when run on one processor. Moreover, *fft/crl* results in about 30% more network traffic than *fft/sc*.

Blocked LU Decomposition The performance of *lu* provides several interesting observations. The overhead of software cache coherence on small 4×4 blocks of 128 bytes overshadows any benefits of caching. Additional measurements show that bigger blocks allow less frequent and larger data transfers, which on high-bandwidth machines closes the gap between *lu/crl* and *lu/sc*. Even so, for 16×16 blocks of 2048 bytes, *lu/sc* performs 1.25–1.4 times better than *lu/crl* on RMC1, RMC2 and the SP2. Apart from the caching overhead in CRL, two factors are responsible for this remaining difference: *lu/sc* uses one-way stores and it separates the communication phase of the program from the computation phase by obtaining all remote blocks before updating local parts of the matrix. In contrast, *lu/crl* interleaves computing and data transfers.

Figure 5 demonstrates the impact of using push-based communication using Split-C's one-way split-phase stores. The effect is quantified by comparing *lu/sc* with *lu-pull/sc* which uses `bulk_get`s instead of `bulk_store`s. On the SP2 performance decreases by about 3%, while on the Meiko performance decreases by about 18%. It is important to note that the computation in which one-way stores were replaced by request/reply constructs only accounts for about 19–26% of the running time of *lu*. This means that other applications may see a substantially larger degradation.

On the other hand, *lu-pull/sc* performs slightly better than *lu/sc* on RMC1 and RMC2. This occurs because RMC1 and RMC2 support remote memory access (PUT/GET) directly in hardware. Split-C's `bulk_get` translates directly into RMC1/RMC2 hardware primitives but `bulk_store` doesn't. `bulk_get` issues a hardware GET operation and executes a local completion handler. In contrast, `bulk_store` issues a hardware PUT operation followed by a message to schedule a handler on the destination node. This asynchronous scheduling of a remote handler is not directly supported in hardware and causes `bulk_store` to lose its advantage over `bulk_get` on RMC1 and RMC2.

Separating communication from computation is motivated in Split-C by the fact that most Active Message implementations are polling-based. It is a common idiom to first gather all necessary data, synchronize globally, and then compute locally. The synchronization avoids conditions in which a slow data requestor is serviced only after the owner of the data finishes its computation, leading to long waits. *lu-c/sc* highlights the impact of this methodology by interleaving computation and communication in a part of *lu/sc* which accounts for about 62-69% of total execution time. On the SP2 *lu-c/sc* runs 1.4 times slower and on the the Meiko the difference is almost three-fold. The fact that the communication latency is not longer masked by issuing multiple split-get requests compounds the issue.

To determine whether the same methodology improves the CRL version we implemented *lu-pref/crl*. Like *lu-pull/sc*, this version reads all blocks that are needed before executing the main computation loops. Indeed, it performs better than *lu/crl* on all platforms, but the blocking nature of the CRL remote access primitives limits the benefit.

Water *water/sc* uses small atomic messages to read from and write to the remote molecules and outperforms *water/crl* on the SP2, RMC1, and Meiko. The CRL network times are considerably higher than using Split-C due to the fact that cached CRL region size does not match the actual amount of shared data, causing CRL to transfer a large amount of unused data. CRL does benefit from caching during the read phase of each time step, but not enough to compensate for the time wasted in data transmission. However, in RMC2, *water/sc* performs slightly worse than *water/crl*. The high network latency in RMC2 hurts the round-trip time of small messages and justifies data caching in *water/crl*.

The performance of *water/sc* can be further improved by replacing the atomic read requests with selective prefetching (*water-slpf/sc*). Additional measurements show that (i) *water-slpf/sc* transfers about 10 times less data than *water/crl* and that (ii) the running time of *water-slpf/sc* is reduced by as much as 44% on RMC1, 64% on the SP2, and 73% on RMC2. Unlike *water/sc*, *water-slpf/sc* no longer pays for the overhead of issuing small messages, which has larger performance impact on platforms with high network latencies (SP2 and RMC2). These observations demonstrate that selectively gathering and explicit bulk copying shared-data are paramount for achieving better performance in Water.

5 Related Work

As far as we know, this is the first study that compares and evaluates the performance of caching, bulk communication, split-phase communication and push-based communication in all-software global address space systems. Existing all-software systems provide a subset of the four communication mechanisms, and previous research on such systems usually evaluate the systems either by comparing them to an all-hardware implementation of shared memory, or by demonstrating acceptable application speedups. Other research investigates the benefits of the individual communication mechanisms in isolation.

Previous research clearly shows the benefits of bulk communication. In a simulation study, Chandra and Larus [3] find that bulk transfer in message-passing systems yield a significant performance advantage over shared-memory systems. Studies on adding message passing primitives to shared-memory architectures also confirm the benefits of bulk transfers [11, 20]. Lui *et al.* [13] study the performance of TreadMarks, a page-based mostly-software DSM system, and find that TreadMarks' inability to combine data on different pages into a single bulk transfer impacts performance negatively. These studies also reveal that DSM systems incur additional communication overhead that is proportional to the amount of cache/page/region misses and protocol messages when transferring large amounts of data. This supports our finding that caching small regions of data in CRL hurts performance.

The benefits of split-phase communication for overlapping communication and computation has been demonstrated both analytically and experimentally in the LogP model [6] and with Active Messages [19].

Scales and Lam [16] demonstrate the benefits of caching and push-based communication when evaluating the SAM shared-

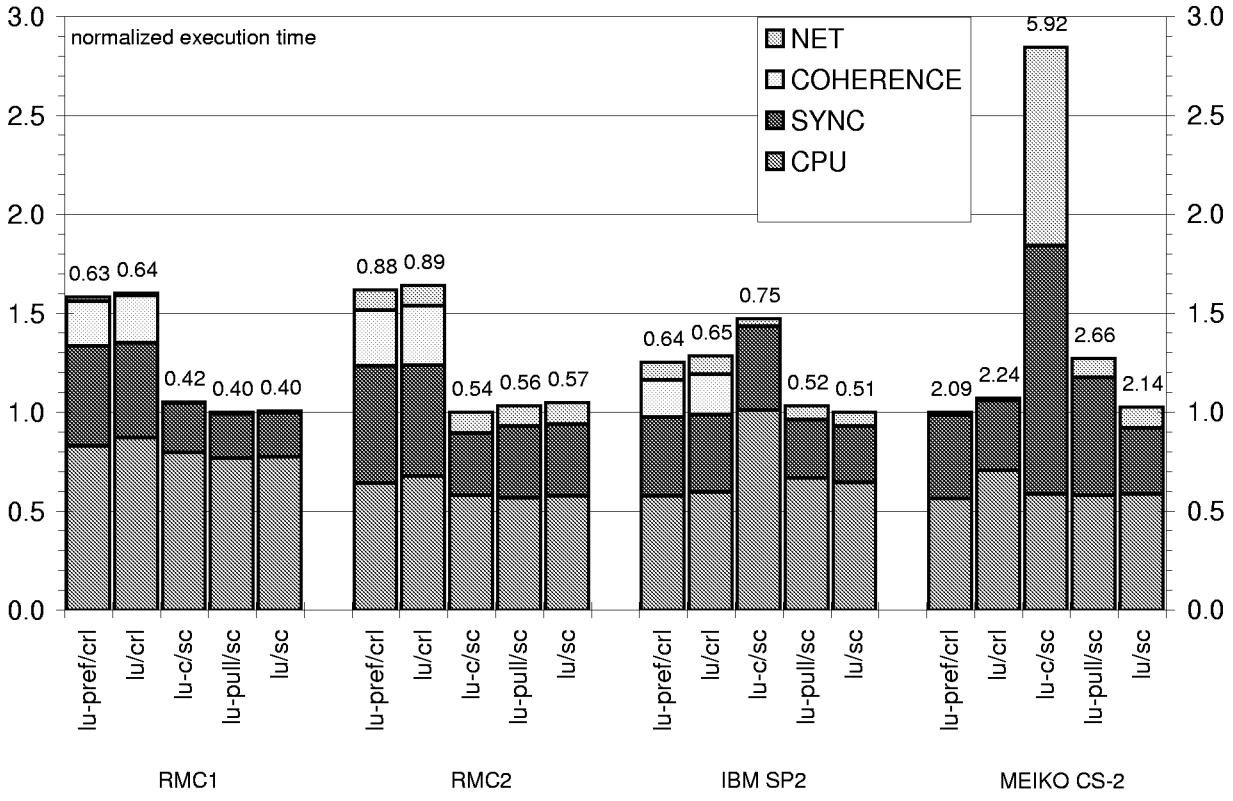

Figure 5: Relative execution times for several versions of *lu* on four platforms. The absolute execution time in seconds is shown above each bar.

object programming system. They selectively turn off caching and push-based communication and show that caching yields tremendous performance improvements (of up to 62 times) and that push-based communication yields a further improvement of up to 31%. However, they do not quantify the effect of using push-based communication without caching.

6 Conclusions

This paper investigates the performance implications of communication mechanisms in all-software global address space systems to tolerate latencies and overheads: caching, bulk communication, split-phase communication, and push-based (sender-initiated) communication.

The results show that bulk communication, either through large CRL regions or explicit bulk Split-C data copies, is essential for achieving good performance in most of the applications we considered. Gathering non-contiguous data to aggregate data transfers improves performance further. Push-based communication improves performance for machines that do not have hardware support for one-sided remote memory access. Caching helps irregularly structured applications and applications with sufficient temporal locality, especially under high network latencies. However, caching also hurts performance when there is not sufficient data reuse, regions are too small or when the region size exceeds the actual amount of data used.

In our experience, the programming complexity of using the mechanisms in CRL and Split-C is comparable. For CRL-style region caching, the main complexity lies in partitioning the program data structures into regions of appropriate sizes that are large enough for performance, but not so large as to cause unnecessary traffic or false sharing. For Split-C, the main complexity lies in in maintaining local copies of remote data and making correct use of synchronization operations after split-phase operations.

The experimental results and analysis show that an ideal system facilitating high performance computing should provide all four communication mechanisms. This observation is valid for modern state-of-the-art supercomputers like the IBM SP2 and Meiko CS-2, and for hypothetical machines representing future design points with different communication models and network latencies.

7 Acknowledgments

We wish to thank Kirk Johnson for help with CRL, Eric Anderson for help with the Split-C version of Barnes-Hut, and Klaus Schauser for providing us access to the Meiko CS-2 at UCSB (acquired through NSF CISE Infrastructure Grant CDA-9216202).

References

[1] T. Agerwala, J. L. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. SP2 System Architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[2] 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.

[3] S. Chandra, J. Larus, and A. Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, October 1994.

[4] K. M. Chandy and C. Kesselman. CC++: A Declarative Concurrent Object-Oriented Programming Notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[5] C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP2. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996. IEEE.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, May 1993.

[7] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumeta, and T. von Eicken. Introduction to Split-C. In *Proceedings of Supercomputing '93*, 1993.

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

[9] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proceedings of Hot Interconnects*, August 1993.

[10] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[11] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 54–63, San Diego, May 1993.

[12] B.-H. Lim, P. Heidelberger, P. Pattnaik, and M. Snir. Message Proxies for Efficient, Protected Communication on SMP Clusters. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, San Antonio, TX, February 1997. IEEE.

[13] H. Lui, S. Dwarkadas, A. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings of Supercomputing '95*, San Diego, CA, 1995. ACM.

[14] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A Portable 'Shared-Memory' Programming Model for Distributed Memory Computers. In *Proceedings of Supercomputing '94*, pages 340–349, Washington, DC, 1994. IEEE.

[15] E. Rothberg, J. P. Singh, and A. Gupta. Working Sets, Cache Sizes and Node Granularity Issues for Large-Scale Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

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

[17] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[18] J. P. Singh, A. Gupta, and J. L. Hennessy. Implications of Hierarchical N-Body Techniques for Multiprocessor Architecture. In *ACM Transactions on Computer Systems*, May 1995.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium in Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[20] S. Woo, J. P. Singh, and J. Hennessy. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, October 1994.

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