Unify: A Scalable Approach to Multicomputer Design

James Griffioen, Rajendra Yavatkar, Raphael Finkel

Department of Computer Science
University of Kentucky
Lexington, KY 40506-0027
{griff,raj,raphael}@cs.uky.edu*

May 1995

Abstract

The Unify project is exploring scalable approaches for designing large-scale multicomputers that span large geographical areas. Such multicomputers must operate in an environment characterized by long latencies, network congestion, significant communication errors, and a variety of processors differing in computational power. Typically, message passing systems have been proposed as the only efficient platforms for such an environment. Unify takes a new approach in that it provides efficient and scalable communication, but yet constructs a shared memory programming model where communication details are hidden.

Unify achieves scalability beyond that of existing DSM systems via new data sharing abstractions, scalable synchronization primitives, application specific memory consistency, and efficient multicast protocols. This paper outlines the Unify design and presents experimental results taken from a prototype system. The results clearly demonstrate the superior scalability of Unify's design and also show that even for smaller scale systems, conventional DSM designs execute 21% slower than Unify.

1 Introduction

Advances in computing and networking technologies have created challenging opportunities for research in high-performance distributed computing. The widespread availability of high-performance uniprocessor and multiprocessor workstations combined with evolving gigabit network technologies allows one to envision a massively parallel distributed multicomputer consisting of thousands of workstations and high-performance multiprocessors scattered across a wide area network.

Unfortunately, this new geographic distribution of processing power, memory, and secondary storage has complicated, rather than simplified, the programming and execution environment. Distributed Shared Memory (DSM) paradigms partially solve the problem of dealing with distributed resources by providing what is widely accepted to be a more convenient programming abstraction than that provided by message passing systems. However, the excessive performance costs typically associated with DSM systems have historically turned users to message passing systems where the distribution of resources is highly visible and left to the user to manage [Che84, MvRT+90, OCD+88, FSC89, Ras86, FH88, Sun90].

*This research was supported in part by the National Science Foundation Grant No. NCR-9111323, Grant No. CCR-9309176, and Grant no. STI-9108764.
Several existing distributed systems such as Ivy [LH89], Clouds [RK91], Mirage [FP89], Amber [JAL+89], Munin [CBZ91], and Midway [BZ93] support shared memory abstractions. However, none of these systems directly address the problems associated with supporting distributed shared memory in a large geographically distributed multicomputer spanning both wide area and local area networks with high latencies, varying bandwidth, and congested links. To achieve scalability in a large-scale, loosely-coupled, distributed system we must develop new shared memory abstractions and mechanisms that (1) mask the distribution of resources, (2) limit/reduce the frequency of communication and the amount of data transferred, (3) hide the propagation latencies typical of large-scale networks (e.g., by overlapping communication with computation), and (4) support large-scale concurrency via synchronization and consistency primitives free of serial bottlenecks.

The goal of the Unify project is to design a highly scalable shared memory multicomputer. Ideally the system must provide convenient data sharing abstractions and must do so with performance and scalability similar to that of existing large-scale message passing multicomputers such as PVM [Sun90]. To achieve this, Unify takes an innovative approach to the overall design of distributed shared memory systems. Briefly stated, Unify’s salient features include:

**Single Address Space:** A single virtual address space, shared by all applications, allows applications to conveniently and efficiently share structured address-dependent data (such as trees and linked lists) as well as other address-independent data.

**Multiple Memory Types:** Unify supports three basic memory abstractions used to store shared data. *RAM* memory is directly addressable. *Sequential access* memory is accessed in a front-read/tail-write fashion. *Associative* memory is accessed via <key,value> pairs. Sequential access and associative memory can often be supported with weaker spatial consistency guarantees.

**Multiple Grades of Consistency:** Unify supplies a set of consistency management primitives that allow an application to select the appropriate consistency semantics from a spectrum of consistency protocols, including *automatic* methods in which the operating system enforces consistency and *application-aided* methods in which the user defines consistency checkpoints. Unify also introduces a new consistency dimension called *spatial* consistency. Spatial consistency determines the relative order of the contents of the replicas of a segment. For many distributed applications that use keyed lookups or sequential access, the order of the data items within a segment is unimportant; only the values of individual data items are important. Spatial consistency allows efficient implementation of such applications.

**Scalable Synchronization Primitives:** Most DSM designs support *locks, semaphores, and/or barriers* as the basic synchronization methods. However, these synchronization primitives pose a serious bottleneck, particularly for large systems with high latencies. To provide efficient synchronization in the presence of long and varying latencies, Unify provides support for a modified form of *eventcounts and sequencers* [RK79]. For a large class of applications, eventcounts result in reduced communication and greater concurrency.

**Hierarchy of Sharing Domains:** To exploit localized sharing and to improve scalability, Unify partitions the set of hosts into *sharing domains*. Each sharing domain uses a separate multicast group to reduce the cost of intra-domain information sharing. Sharing domains distribute the burden of information retrieval and distribution by allowing any member of the domain to issue or answer inter-domain requests (addressed to multicast groups). Every host consults the hosts
in its local sharing domain before going outside the domain for information. Therefore, as soon as one host obtains cross-domain information from a site in a remote domain, the information effectively becomes available to all other hosts in the sharing domain.

**Reliable Multicast Support.** To achieve reliable, scalable, efficient dissemination of shared data or synchronization information, Unify supports reliable multicast via a tree-based multicast transport protocol (TMTP) [YG95]. TMTP builds on the efficient delivery of IP multicast [DC90] but uses a separate control tree to handle error and flow control resulting in local retransmissions and acknowledgements.

We have implemented a prototype version of the Unify kernel as a Unix user-level library that can be linked with distributed shared memory applications. Section 5 illustrates Unify's superior scalability and performance by comparing the performance of two versions of SOR (one using Unify's new memory abstractions, and one using recent competing DSM abstractions) executing on machines spanning two distinct networks. Our tests successfully demonstrate the scalability of Unify primitives even with 16 nodes in two sharing domains. We observed that the standard DSM version encounters a communication bottleneck (and performance degradation) with only 8 nodes, while Unify's performance continued to improve with as many as 16 nodes. Albeit on a limited scale, our results show that scalable DSM systems can be designed by taking an innovative approach to the problem of consistency management and synchronization.

# 2 Unify: An Overview

The Unify multicomputer consists of a large number of nodes (workstations or small multiprocessors) each executing the Unify distributed operating system. Unify integrates this large set of independent machines into a whole that resembles a single, large, shared memory multiprocessor.

## 2.1 The Architecture

We believe that sharing in a large scale distributed multicomputer will follow the principle of locality. That is, a given Unify host will share large amounts of data with nearby sites, considerably less with more distant sites, and almost no data with very remote sites. In hope of exploiting this locality, Unify partitions the set of hosts into *sharing domains*. Each sharing domain contains a set of machines likely to share substantial amounts of data interconnected by a relatively low-latency, high-bandwidth communication network such as a common bus, an Ethernet, or a fiber-optic net. Network connections among sharing domains may provide high-bandwidth but latency will be considerably higher than intra-domain transfers.

This architectural organization allows Unify to optimize for the most common forms of data sharing. In addition, sharing domains can substantially reduce network traffic when information is shared across domains by batching multiple inter-domain sharing requests.

---

¹Many potential large-scale distributed applications, particularly numerical applications, can be designed in such a way that they exhibit this behavior. Certainly applications exist that violate our assumption and share large amounts of data among many, if not all, machines. Often, such applications are simply not well-suited for a large-scale distributed environment, be it DSM or message passing. Despite the inappropriate nature of such applications, Unify's sharing domains and multicast protocol can actually outperform message passing implementations.
2.2 Unify’s Process Structure and Memory Organization

Unlike most conventional virtual memory systems, Unify supports a single, large, virtual address space\(^2\). The global virtual address space is shared across all Unify nodes, facilitating easy data sharing among processes regardless of their location. All computations throughout the entire multicomputer execute in this single large address space.

Figure 1 shows the memory space organization in Unify. A *segment* is a contiguous region of the global virtual address space into which data can be stored or retrieved. Each segment is defined by a starting address, a length, and a set of attributes. Sharing is at the granularity of segments. Unify further organizes the global address space by grouping segments into clusters and workspaces.

A *cluster* defines a group of related segments that the operating system (and applications) can conveniently manipulate as a single logical entity. The individual segments that comprise a cluster may have different attributes thereby allowing the individual components of a cluster to behave differently depending on their use. Clusters arise naturally in many applications. For example, consider an object-oriented application where each object contains two segments: a methods (code) segment and a data segment.

Finally, a *workspace* defines the set of clusters (and thereby segments) visible to a particular computation and is similar to a process’ virtual address space in conventional Unix systems. The basic unit of computation in Unify is called a *worker*. At most one worker inhabits each workspace. For the purposes of this paper, we will use the terms worker, process, and thread interchangeably.

![Diagram of Unify's memory structure](image)

Figure 1: Two views of the memory model in Unify. The left view shows the logical organization, while the right view shows the arrangement of the segments in the global address space.

### 2.2.1 Memory Types

Unify supports three types of segments: *random access segments*, *associative segments*, and *sequential access segments*. Conventional DSM systems only support random access memory regions. Unify’s

\(^2\)Recent hardware advances have made very large virtual address spaces possible. Several architectures and operating systems now provide 64-bit address spaces capable of holding vast amounts of data [CLLBl92, GBK92].
two additional segment types serve two purposes. First, many programming constructs/abstractions can be more easily programmed using these memory abstractions. Second, sequential access and associative segments often scale better and provide better performance than random access segments. Supporting multiple memory types allows the application to choose the most appropriate storage mechanism for the task. These segment types allow direct, keyed, and "read-front, write-append" access as described below:

**Random Access:** Read/write operations on random access segments return/modify the value at a particular address in the segment. This is the conventional memory abstraction (RAM).

**Sequential Access:** A sequential access segment is accessed in a read-front, write-append manner. Unify associates a per-workspace `nextRead` pointer with the segment to keep track of the next location within the segment to be accessed by a read operation. Thus, `read(SeqSeg)` returns the contents of the location at the current workspace's `nextRead` address. If no new value is available, the caller is blocked until a new value becomes available. `Write(SeqSeg, value)` atomically appends the given value to the end of the segment, represented by the globally shared pointer (`nextWrite`). Sequential access segments are particularly useful for applications such as distributed logs that are accessed in a sequential manner. Sharing overhead can be greatly reduced if the entries need not appear in the same order at all replicas. Furthermore, `nextRead` and `nextWrite` operations provide implicit synchronization between a reader and a writer to the same segment.

**Associative:** An associative segment supports associative access and is organized as a collection of `<key, value>` pairs. Associative segments impose no ordering among the pairs within the segment. Associative segments are useful for distributed directory services such as name servers that support associative lookup on shared depositories of data.

3 Application-specific Consistency Models

Like other DSM systems, Unify must address the *cache coherency* problem [Smi82]. However, unlike existing distributed shared memory systems, Unify provides a new perspective on consistency by defining consistency of a shared segment across two dimensions (time and space) rather than just one dimension (time) as is usually done. We refer to consistency in the time dimension as *temporal consistency* and consistency in the space dimension as *spatial consistency*.

**Temporal consistency** is the classical way of viewing consistency. When a shared segment is modified concurrently at several sites, temporal consistency determines when (and in what order) changes to a shared segment are made visible to all the sites interested in a segment. As described below, Unify allows an application to choose (on a per-segment basis) from a variety of temporal consistency semantics.

**Spatial consistency** determines the relative order of the contents of the replicas of a segment. When two copies of the segment contain the same values in the same order we say the segment is *in-order* consistent. When the two copies have the same values but in different orders we say the segments are *out-of-order* consistent. Example applications that can tolerate out-of-order segments include associative memory (tables, name registries), distributed log files, databases, and numerical iterative methods that produce values to be reduced by an associative and a commutative operation.

---

3 Associative segments provide an abstraction similar to the tuple space abstraction supported in Linda [GCCCS86, Gel89].
The following sections describe the semantics of both temporal and spatial consistency in Unify.

3.1 Temporal Consistency

Temporal consistency methods in Unify fall into one of two categories: application-aided consistency and automatic consistency.

Application-aided consistency methods allow the application to specify the points in the code where the cached copies need to be consistent. These methods capitalize on the fact that shared memory programs often use synchronization variables to guard access to shared data and, therefore, updates to shared data can be delayed until the end (or beginning) of a guarded region. In addition, these methods batch multiple updates in a single message to further improve performance. For example, under release consistency [GLL+90, CBZ91], updates to shared data are not propagated to other cached copies until the end of a critical section (when a lock is released)\(^4\). Unify provides a set of consistency management primitives and operations that allow an application to delay updates to shared data and to retrieve the most recent version of shared data on demand. Using these primitives, an application can achieve the effect of consistency protocols such as weak consistency [DSB86], release consistency [GLL+90, CBZ91], buffered consistency [SR92], or entry consistency [BZS93]. Further details can be found in [GYF94].

Under Unify’s automatic consistency methods, the system automatically and transparently brings cached copies of a segment into a consistent state. For instance, Unify supports sequential consistency [Lam79] where updates are propagated immediately, before any other operation can be performed on the segment. However, such guarantees are often too strict, and are thus more expensive than necessary. Therefore, Unify also provides best-effort consistency and eventual consistency. Like sequential consistency, these two methods automatically ensure that cached copies contain up-to-date data. However, the two methods differ from sequential consistency in the amount of time \(t\) it takes for the method to ensure all replicas have up-to-date information after a change. Unify allows an application to specify the value \(t\) for the automatic update period on a per-segment basis.

As opposed to sequential consistency which guarantees that updates will happen immediately (\(t\) is zero), best effort consistency makes no such guarantees. Essentially the value of the update period \(t\) is unknown and may even be infinity. Under best-effort consistency, each site is to free to propagate its updates any time and typically would transmit them periodically. However, Unify neither guarantees that all sites receive all updates nor requires that all sites reach a stable state. At first, such unreliability may appear useless. However, best-effort consistency proves beneficial to a wide variety of distributed applications that can detect stale data or detect (and even correct) inconsistencies using problem-specific semantics. Examples of such applications include ARP caches [Plu82], distributed routing update protocols, and the Domain Name System (DNS) [Moc87].

Eventual consistency \((t = \text{some known value})\) lies between sequential consistency and best-effort consistency. Eventual consistency requires that each site periodically (with period \(t\)) transmit updates made to shared data. Unify guarantees that all cached copies receive all updates. If two sites concurrently generate conflicting updates, workers must use synchronization mechanisms to prevent such conflicts. Because updates are transmitted every \(t\) time units, the replicas will eventually reach a mutually consistent state in the absence of new changes to the shared data.

Eventual consistency typically results in a substantial performance improvement over sequential consistency and suffices for a wide variety of distributed applications. Moreover, many distributed applications can tolerate temporary inconsistencies [Che86]. Systems such as Grapevine [BLNS82]

\(^4\)We have used release consistency in our prototype implementation described in Section 5
only require that shared data eventually reach a consistent state if no new updates occur. Several other examples of distributed applications with similar semantics can also be found in the literature [Che86, ASHP87, Lam83].

3.2 Spatial Consistency

Spatial consistency determines the relative order of the contents of the replicas of a segment. We say that the cached copies of a shared segment are \textit{in-order consistent} if the order of the data within the segment is the same at all sites. Similarly, if all the cached copies contain the same information, but in a different order, we say the cached copies are \textit{out-of-order consistent}.

For example, consider a log segment with 100 log entries that is cached at three sites: A, B, and C. Suppose a worker at site A appends an entry to the log, and at the same time a worker at site C also appends an entry to the log. Both A and C propagate their change to all the other sites. The replicas are \textit{in-order consistent} if all the log entries in the segment appear in the same relative order at all three sites. The replicas are \textit{out-of-order consistent} if the cached copies at each site contain the same log entries but the relative order of the entries within the segment differ (for example, the entries at site A are ordered differently than the entries at site B or C).

Most shared-memory models only provide \textit{random} access memory. Consequently, any consistency guarantee other than in-order consistency (e.g., out-of-order consistency) is of no value. However, out-of-order consistency is meaningful under Unify's sequential access and associative access memory types and can substantially improve performance. The relative ordering of values in an associative memory is of no interest to the application and the relative ordering of values in a sequential access memory segment often turns out to be irrelevant. For example, consider a parallel implementation of a numerical iterative method in which many workers write new data values to a shared segment at the end of each iteration. These values are then read by one or more workers for further processing. Because each entry in such a segment can contain ordering information (which iteration, which value), there is no need to expend extra effort to insure that all entries appear in the same order across all cached copies.

In Unify, associative segments use out-of-order consistency. In the case of sequential access segments, however, an application may specify in-order or out-of-order consistency. Random access segments are always in-order consistent. The out-of-order consistency model provides a significant performance advantage for many distributed applications and still allows them to choose the appropriate temporal consistency guarantee.

4 Synchronization

Synchronization in Unify sets it apart from all other DSM models. Other distributed shared memory models provide some combination of locks, semaphores, spinlocks, monitors/condition variables, or barriers [GLL+90, CBZ91, BZS93, SR92, CAL+89] as the synchronization mechanisms. Although these mechanisms can be easily and reasonably efficiently implemented on multiprocessor architectures, some even in hardware, they can, and often do, become a serious bottleneck to the performance of distributed multicomputers. For instance, spinning is an expensive operation in a loosely coupled multicomputer with an underlying unreliable communication facility. Despite the potential bottlenecks, existing distributed shared memory systems have resorted to these same basic synchronization techniques for various reasons (e.g., to ease the port from multiprocessor codes).

To obtain scalable synchronization primitives in a wide-area multicomputer requires a new look
at the synchronization requirements of applications. Note that all of the above synchronization abstractions make the underlying assumption that synchronization accesses must be sequentially consistent. That is, all sites must observe the same set of synchronization accesses in the same order. Given the high latencies and error rates of large networks, such restrictions can be very costly. Although some applications may require this strict form of synchronization, many applications do not rely on ordered synchronization access and can realize significant performance benefits from weaker synchronization constraints.

The key to supporting scalable synchronization accesses is the abstraction of *eventcounts* [RK79]. The operations associated with eventcounts do not care when an event occurred, only that the event occurred. Consequently, the ordering of synchronization events is irrelevant.

For example, consider the very common "single-producer/single-consumer" situation. In this case, the consumer does not need to know exactly "when" a particular slot in the shared bounded buffer was filled; rather, the consumer is only interested in the fact "that" it was filled. Consequently, the system need not expend unnecessary effort to immediately inform the consumer of synchronization events executed by the producer.

Unify provides three operations for manipulating eventcounts: *Advance(eventcount)* increments the value of an eventcount, *Wait_with_timeout(eventcount, count, timeout)* waits for the value of an eventcount to reach or surpass the value *count* within a fixed time interval *timeout*, and *Event_sync(eventcount)* insures that the local value of an eventcount is up-to-date. These operations differ from the standard notion of eventcounts in that they allow timed waiting, consistency checkpoints, and avoid serialization and communication bottlenecks. For instance, the *Advance* operation uses unreliable (IP/UDP) multicast to propagate the increment to all the sites and relies on a site to use *Event_sync* if it misses some or all of the updates.

If applications depend on ordered synchronization events, Unify supports the sequencer [RK79] abstraction which allows applications to achieve a unique, global ordering of events through its *get_ticket* operation. In addition, Unify supports a *get_r_ticket(r_sequencer)* operation that returns an arbitrary unique number. The set of numbers returned is not guaranteed to be either dense (there may be unused numbers) or sorted (numbers can be given out in any order). An efficient implementation of this operation divides the sequence number space among multiple sites, with each site in charge of handing out its set of sequence numbers. Thus, *get_r_ticket* avoids the distributed mutual exclusion necessary with the ordinary sequencer, yet provides a unique ticket mechanism which is useful in a variety of applications.

5 Experimental Results

We have implemented the Unify system as a runtime library on Unix workstations. DSM applications link with the library to create shared segments with appropriate memory types and consistency semantics.

To achieve a testbed environment that, to some extent, resembles a realistic distributed system, our current tests use a set of 16 non-identical workstations scattered across two distinct 10 Mbps Ethertnets connected by two gateways. The 16 Sun SPARC-based workstations differed in their processing speed (25-50 MIPS range), memory size, and load (some act as file servers). In addition, network latencies and packet loss rates varied depending on whether communication was local or across the gateway.

To illustrate the advantages of sequential access segments, we experiment with two different versions of Successive Over Relaxation (SOR); one using random access segments and one using
sequential access segments. During each iteration, the new value at each grid point is calculated by computing the average of its four “neighbors”. At the end of each iteration, adjacent workers exchange computed values at section boundaries.

The (conventional) random segment version used random access segments for the private elements and for the shared peripheral elements. “Shadow” segments were used to avoid overwriting old values before they were used. The peripheral random access segments were declared as release consistent [GLL+90, CBZ91] so that updates would only be propagated at the end of each iteration. Barriers were used to synchronize the processes at the end of every iteration. In short, the random access version tries to incorporate the best ideas of the modern approaches mentioned in section 3.

The sequential segment version also used a random access segment (and shadow copy) for the private elements. However, sequential access segments were used to hold peripheral elements. Like the random version, the peripheral segments were declared as release consistent to delay updates until the end of an iteration. However, synchronization was implicit via the sequential access segment.

We measured the performance of the random and sequential access versions, each processing 100 iterations on a 1024 x 1024 matrix during evening hours. The results are shown in Figure 2. For small numbers of hosts (2-6), the performance is roughly the same. However, already at 6 hosts, the random version runs 21% longer than the sequential version. With as few as 8 hosts, the random version begins to experience heavy packet loss and network congestion (primarily at the gateways) due to the lockstep synchronization and update exchange at the end of every iteration (which does not occur in the sequential access version due to the implicit synchronization afforded by the read_next operation). Overall, even the best (up to 16 host) performance of the random version is 66% slower than that of the best sequential version.

6 Conclusions

We have described the design of the Unify distributed shared memory system and discussed how new memory abstractions, consistency, and synchronization mechanisms can be combined to offer
scalable solutions to the problem of building a large scale distributed shared memory multiprocessor. Contrary to widely held beliefs, our experimental results show that distributed shared memory systems are not inherently non-scalable or plagued by high overheads. Rather, these problems result from conventional sharing abstractions and implementations. In particular, our tests show that sequential shared segments can result in as much as a 66% performance improvement over a conventional random segment implementation.

7 Acknowledgments

We would like to thank Chris Diaz for helping with the implementation of the Unify library and example applications.

References


11