Unify: A scalable, loosely-coupled, distributed shared memory multicomputer

Technical Report 226-93

James Griffioen
Rajendra Yavatkar*
Raphael Finkel

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

January 31, 1993

Abstract

Unify is a scalable approach for designing distributed multicomputers. It uses high-speed local and wide area networks as a distributed backplane to interconnect hundreds or thousands of workstations and supercomputers into a single massively parallel multicomputer. From a programmer's point of view, Unify's memory model provides a high-performance shared memory multiprocessor, making it easy to program massively parallel applications. Unify achieves scalability beyond that of existing distributed shared memory approaches by providing scalable synchronization primitives, multiple grades of consistency for shared memory, lazy evaluation for kernel data structures, optimizations for local sharing, and efficient multicast protocols. This paper describes the Unify design and discusses the scalability issues involved in constructing a large scale distributed shared memory multicomputer.

*This research was supported in part by the National Science Foundation Grant No. NCR-9111323 and Grant no. STI-9108764.
1 Introduction

Advances in computing and networking technologies have created challenging opportunities for research in high performance computing. As workstations get faster, supercomputing technologies evolve further, and gigabit network technologies become available, we can now envision a massively parallel distributed multicomputer consisting of thousands of workstations and supercomputers scattered across a wide area network.

However, we must address several research issues before we can realize such a multicomputer. The geographic distribution of processing power, memory, and secondary storage has complicated, rather than simplified, the programming and execution environment. To make effective use of such an environment, we must develop programming abstractions that mask the distribution of resources and facilitate the construction of massively parallel applications. Moreover, we must design algorithms (for synchronization, resource management, and data exchange) that scale well and deal with propagation latencies across a high-speed network.

Unify is a scalable approach to designing distributed multicomputers. The system uses conventional and emerging high-speed local and wide area networks as a distributed backplane to interconnect many machines into a single, massively parallel computer. A unique distributed shared memory architecture transforms the collection of machines into a massive high-performance shared memory multiprocessor with novel operations that simplify the design of large-scale parallel and distributed applications.

Unify’s chief objective is scalability. Unify achieves scalability beyond that of existing distributed shared memory approaches by providing scalable synchronization primitives, multiple grades of consistency for shared memory, lazy evaluation for kernel data structures, optimizations for local sharing, and efficient multicast protocols.

This paper describes the Unify design and discusses the scalability issues involved in the design of a large scale distributed shared memory multicomputer. We begin by motivating the need for large scale distributed shared memory models. Section 3 describes the scalability issues that affect the design of such a model. Sections 4, 5, and 6 present an overview of Unify followed by a description of Unify’s sharing and synchronization primitives. Section 7 provides a sample parallel application and its implementation on Unify. Section 8 describes related work and compares it with the novel aspects of Unify. Finally, Section 9 describes the current status of Unify and summarizes our contributions.
2 Motivation

It is commonly believed that the communication-kernel approach is the best for building distributed operating systems that span a large multicomputer. Communication kernels, such as V [Che84], Amoeba [MvRT+90], Sprite [OCD+87], Charlotte [FSDL83], and Yackos [FH88], provide message passing between processes executing in disjoint virtual spaces as the basic mechanism for invoking services and sharing information. Dual to the communication-kernel approach is the shared-memory kernel approach [LN78]. Under the shared-memory approach, processes communicate via shared memory (implicit communication) and coordinate their activities using synchronization primitives. Shared memory allows a finer grain of parallelism than the communication-kernel approach. It also provides a convenient basis for migrating processes from one host to another. Moreover, we feel that shared memory provides a more suitable programming model for creating highly parallel applications.

Several existing distributed operating systems support the shared memory abstraction, including Ivy [Li86], Clouds [RK91], Mirage [FP89], Amber [JAL+89], Munin [CBZ91], and Midway [BZ91]. All these operating systems have focused on consistency management or the application-level interfaces to shared memory. None has tackled the problem of scaling to a very large multicomputer spanning wide and local area networks with high latencies and congested links. To achieve scalability the system must avoid serial bottlenecks in synchronization accesses, high latency messages, and algorithms that result in heavy traffic patterns (i.e., congested links).

We have taken an innovative approach in the design of Unify distributed shared memory. The salient features of Unify include:

**Shared Global Address Space.** Unify provides a single global shared virtual address space with unique virtual addresses throughout the multicomputer, allowing applications to conveniently and efficiently share structured address-dependent data (such as trees and linked lists).

**Scalable Synchronization.** Unify provides scalable alternatives for synchronization accesses, unlike existing DSM approaches. Most existing DSM proposals support *locks* and/or *barriers* as the basic synchronization methods. However, these synchronization primitives typically pose a serious bottleneck.

Unify provides support for *eventcounts* as the basic synchronization primitive. In the environment provided by eventcounts, applications do not care *when* an event occurred, they
only need to know that an event occurred. These relaxed constraints significantly improve
the scalability of the model because the data structures associated with the synchronization
primitives need not be kept consistent across all the sites.

For those applications that require a strict (serial) ordering of events, Unify supports a ticket
operation (a sequencer) with cost similar to the cost of a lock operation in a classical dis-
tributed model. The ticket operation maintains a nondecreasing integer variable, initialized
to zero, that can be used to totally order events. However, many applications do not require
a total ordering of events. Therefore, Unify provides a less costly random_ticket operation.

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 ensures
consistency and application-aided methods in which the user defines consistency checkpoints.

Unify also introduces a new consistency dimension called spatial consistency. Spatial con-
 sistency determines the relative order of the contents of the replicas of a segment. In many
distributed applications that use associative memory and distributed logs, 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.

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 multi-
cast 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. There-
fore, 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.

Lazy Evaluation of Kernel Data Structures. To improve scalability, the Unify kernel itself
uses a weak form of consistency to maintain the coherence of many shared kernel data struc-
tures.
3 Multicomputer Scalability Issues

Current research in shared memory multiprocessors has addressed some of the issues in scaling such machines to a large number of processors. However, a multicomputer differs from a multiprocessor (such as a Sequent Symmetry, a Kendall Square Research machine, or a BBN Butterfly) in several key ways that substantially affect the scalability issue.

The first and the most important difference is communication latency between processors, which is several orders of magnitude greater in a multicomputer than in a typical multiprocessor. In addition, latencies vary widely in a multicomputer, while they are of lower variance in a multiprocessor.

A second difference occurs in the way messages are routed between processors. Multicomputers dynamically route messages among nodes, while messages in a multiprocessor typically follow a fixed route with a fixed latency.

Thirdly, all the nodes (processors) in a multiprocessor are typically of equal computing power, and all network links have generally equal bandwidth. However, in a multicomputer, the processor speed and the bandwidth of the interconnection network may vary from node to node and link to link respectively.

Fourthly, multiprocessors often have a globally accessible memory in addition to local caches on the processors. In contrast, each node in a multicomputer can only directly access its local memory. These differences give rise to a new set of constraints from which we identify three general scalability issues that must be addressed by a multicomputer: latency, congestion, and resource allocation.

Latency. Although network bandwidths have increased, communication latency continues to dominate as the size of the interconnection network increases. Latencies on the order of milliseconds or even hundreds of milliseconds are not uncommon. These types of delays can seriously affect the performance of a shared memory application. In order to scale to several hundreds of hosts, the multicomputer must attempt to localize communication and avoid or overlap high latency messages whenever possible.

Congestion. Congestion occurs in two ways. First, centralized algorithms unfairly burden one processor with a large portion of the work, causing the processor to become a bottleneck (processor congestion). Even otherwise distributed algorithms can have single points of serialization that can quickly become a bottleneck [Li86, LH89]. Second, network congestion
may result if distributed algorithms that implement synchronization or consistency involve frequent communication among all the nodes. Consequently, the multicomputer must use efficient methods to prevent congestion and respond to it when it does occur.

**Inequitable Distribution of Resources.** Such large scale systems will inevitably consist of nodes with a wide range of computing power and resources. Managing (allocating) this inequitable set of resources in an optimal fashion becomes increasingly complex as the size of the system increases.

## 4 The Unify Multicomputer: 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. The following sections provide a brief overview of Unify.

### 4.1 The Unify 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. In addition, we expect that the interconnection network linking hosts in a sharing domain provides relatively low-latency, high-bandwidth communication, such as through a common bus, a single Ethernet, or a single fiber-optic net. Network connections between sharing domains may provide high-bandwidth data transfers, but we expect the latency to be considerably higher than the latency within a sharing domain.

This architectural organization allows Unify to optimize for the most common forms of data sharing. Sharing domains reduce network traffic when information is shared across domains by batching multiple inter-domain sharing requests.
4.1.1 Memory Space Organization

The Unify memory model consists of a globally addressable virtual address space. Unlike most conventional virtual memory approaches, Unify supports a single, large, virtual address space\(^1\). The global virtual address space is shared across all Unify nodes. Each may access any location in the shared global address space at any time. All computations throughout the entire multicomputer execute in this single large address space.

Because the shared global address space provides unique virtual addresses throughout the multicomputer, applications can conveniently and efficiently share structured address-dependent data (such as trees or linked lists). Consequently, applications do not need to spend time and code marshalling or unmarshalling data structures.

Unify partitions the global virtual address space into segments. 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. 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 manipulate as a single logical entity. 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. The two segments might have different access restrictions and need not be contiguous. However, logically the two segments belong together. Clustering provides the logical grouping required to conveniently manipulate such related segments.

The workspace abstraction serves as the second level of grouping in the Unify memory organization. Each workspace defines the set of clusters (and thereby segments) visible to a particular computation. A cluster may appear in zero or more workspaces, thereby allowing multiple computations to share the segments in a cluster. Likewise, a segment may appear in zero or more clusters, allowing multiple clusters to map the same segment. To protect a segment against unauthorized access, each segment maintains access rights for each workspace that has access to the segment. Figure 1 shows the memory space organization in Unify.

---

\(^1\)Recent hardware advances have made very large virtual address spaces possible. Several conventional architectures now provide 64-bit address spaces capable of holding vast amounts of data [CLBHIL92,GBK+92,Cor91]. Although Unify assumes a large global address space, a small global address space (for example, a 32-bit address space) would only affect its scalability, not its functionality or performance.
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.
4.1.2 Segment Types

Unify supports three types of segments: random access segments, associative segments, and sequential segments. As the names imply, each segment allows direct, keyed, and front-read, end-write access respectively. By supporting multiple segment types, Unify allows applications to choose the most appropriate storage mechanism for the task. We describe each of these types and their associated semantics below.

Random access. Read/write operations on random access segments provide the same semantics (direct access) as conventional random access memory (RAM). The operation read returns the contents of a given virtual address, and the operation write writes the given value to the virtual address specified. Programs will typically use segments of this type to manipulate parts of their address space in a conventional manner.

Sequential. A sequential segment is only accessed in a sequential manner. When a segment of this type is included in a workspace, Unify associates a next_read pointer with the segment to keep track of the next location within the segment to be accessed in a read operation on a per-workspace basis. Thus, \texttt{read(SeqSeg)} returns the contents of the location at the current workspace's next_read address. \texttt{Write(SeqSeg, value)} atomically writes the given value to the last location (next_write) in the segment. Unlike Next_read, Next_write is effectively a global pointer shared by all workspaces.

Sequential segments are useful for applications such as distributed logs that are accessed in sequential manner. Sharing such segments across multiple sites is easy if processes at each site only demand an up-to-date segment view that contains all the log entries even if the same log entries may appear in different order on different sites.

Associative. An associative segment supports associative access and is organized as a collection of \texttt{<key, value>} pairs. No order is imposed on the contents of the segment; both key and value fields are of fixed size. Read and write operations for such segments require a key as an argument.

Associative segments are useful for distributed directory services such as name servers that support associative lookup on shared depositories of data.
4.2 The Computational Model

The basic unit of computation in Unify is called a worker. Workers represent the active entities in Unify and can be viewed as concurrently executing threads of control. Unlike multi-threaded processes in conventional operating systems (such as Mach [Ras86], Psyche [SLM90], and Chorus [AR89]), each worker in Unify executes in its own workspace. That is, each workspace in Unify supports at most one worker. However, because a cluster can appear in multiple workspaces, we can easily achieve the multi-threaded process model by having multiple workers map the same clusters.

The Unify process model provides a great deal of flexibility. Each worker can dynamically alter its workspace by mapping clusters into the workspace or unmapping clusters from the workspace. Because all workers execute in a single global address space, workers can freely share arbitrary portions of their data. Synchronization primitives allow multiple workers to coordinate their activities.

5 Application-specific Consistency Models

Unify treats the physical memory at each site as a local cache that holds copies of the segments currently needed by workers at the site. Because each node's memory acts solely as a cache, Unify must address the cache consistency problem [Smi82, Li86]. If one site writes to its local cached copy, it must also propagate the change to all other sites to make all replicas consistent. Previous research in distributed shared memory has produced a variety of methods for bringing replicas into a consistent state. These methods provide a variety of consistency guarantees ranging from strict consistency (for example, sequential consistency as in IVY [Li86] or Mirage [FP89]) to weaker forms of consistency such as release (DASH [GLL+90] and Munin [CBZ91]), entry (Midway [BZ91]), or buffered consistency (Beehive [SR92]).

The consistency model in Unify is based on the observation that every distributed (parallel) application has its own application-specific consistency requirements. To provide a convenient programming environment and to achieve superior performance and scalability for a wide range of applications, the shared memory model must provide a variety of consistency guarantees.\(^2\)

Unify's consistency model is innovative in several ways. First, Unify supports a wide range of consistency guarantees. Second, unlike existing distributed shared memory systems, Unify defines consistency of a shared segment across two dimensions (time and space) rather than just one.

\(^2\)Clouds [AAL92] makes a similar observation but arrives at a different conclusion. Section 8 highlights some of the differences.
dimension (time). 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. Unify supplies a set of basic consistency management primitives that allow an application to select (on a per-segment basis) the appropriate consistency semantics from a spectrum of consistency protocols, ranging from *automatic methods* such as sequential consistency to *application-aided methods* such as release consistency.

*Spatial consistency* determines the relative order of the contents of the replicas of a segment. In many distributed applications, the relative order of the data items within a segment is unimportant, only the data items themselves are important. Examples of such applications include associative memory (tables, name registries), distributed log files, databases, and values to be reduced by an associative and a commutative operation.

The following sections describe the semantics of both temporal and spatial consistency in Unify.

### 5.1 Temporal Consistency

Figure 2 shows the temporal consistency models supported in Unify. Temporal consistency methods

<table>
<thead>
<tr>
<th>Automatic Updates</th>
<th>Automatic Consistency Models</th>
<th>User-Aided Consistency Models</th>
<th>User Triggered Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>More Often</td>
<td>Sequential Consistency</td>
<td>Weak Consistency</td>
<td>More frequent</td>
</tr>
<tr>
<td></td>
<td>Eventual Consistency</td>
<td>Release Consistency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Best-effort Consistency</td>
<td>Buffered Consistency</td>
<td>Less frequent</td>
</tr>
<tr>
<td>Less Often</td>
<td></td>
<td>Entry Consistency</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Unify supports two types of temporal consistency: *Automatic* consistency and *Application-aided* consistency.

in Unify fall into one of two categories: *automatic consistency* and *application-aided consistency*. 
Under the automatic consistency methods, Unify automatically and transparently brings cached copies of a segment into a consistent state. When a worker updates a shared data item, all other workers automatically see the change.

*Application-aided consistency* methods allow the application to specify the points in the code where the cached copies need to be consistent. Application-aided consistency capitalizes on the fact that shared memory programs often use synchronization variables to guard access to shared data. Because such access is guarded, the operating system need not expend effort to ensure consistency. Delaying updates to shared data until the end of a guarded region and batching multiple updates in a single message can significantly improve performance.

### 5.1.1 Automatic Consistency Methods

Unify defines three types of automatic consistency: *sequential consistency*, *eventual consistency*, and *best-effort consistency*. In all three cases, Unify automatically ensures that cached copies contain up-to-date data. However, the three methods differ in the amount of time \( t \) it takes for the method to bring all replicas into a consistent state after a change.

Lamport has defined the semantics of *sequential consistency* to mean that the “result of any execution is as if the memory operations at all the sites were executed in some sequential order, and the operations at each site appeared in this sequence in the order specified by the program executing at the site” [Lam79]. Thus, at each access to a shared data item, all the replicas contain identical data and, effectively, the parameter \( t \) (the time elapsed before all replicas are consistent) appears to be zero. Consequently, when \( t \) equals zero, automatic consistency achieves the semantics of sequential consistency.

Although the temporal consistency methods in Unify include support for sequential consistency, the novel aspect of temporal consistency in Unify arises from its support for application-specific settings for the value \( t \), the automatic update period. From this spectrum we identify two new consistency methods: *best-effort* consistency and *eventual* consistency.

*Best-effort* consistency and sequential consistency lie at opposite ends of the spectrum. Sequential consistency guarantees that updates will happen immediately. On the other hand, best-effort consistency makes no guarantees. Under best-effort consistency, \( t \) is unknown and may even be infinity. Under best-effort consistency, each site periodically transmits its updates. However, Unify neither guarantees that all sites receive all updates nor requires that all sites reach a stable state.
Although such unreliability may appear to be useless at first, 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 (Address Resolution Protocol) 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\(^3\) requires that each site periodically transmit updates made to shared data. Unify guarantees that all cached copies receive all updates. Every site must transmit its updates within some fixed period $t$ of making a change. If two sites concurrently generate conflicting updates, Unify does not guarantee consistency. Workers must use synchronization mechanisms to prevent such conflicts. Because updates are transmitted every $t$ time units under eventual consistency, the individual memories will eventually reach a mutually consistent state in the absence of 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. In his paper on problem-oriented shared memory [Che86], Cheriton shows that many distributed applications can function correctly even when cached data is temporarily out-of-date. For example, the Grapevine distributed name server [BLNS82] maintains a name registry replicated across multiple sites. Individual sites maintain cached copies of the registry but do not require sequential consistency between cached copies, because Grapevine clients can tolerate temporary inconsistencies. Grapevine only requires that shared data eventually reach a consistent state if no new updates occur. Several other examples of distributed applications with similar semantics can be found in the literature [Che86, ASHP87, Lam83].

### 5.1.2 Application-aided Consistency Methods

*Application-aided* consistency protocols, as the name implies, allow applications to specify the points at which consistency must occur. These specifications fit naturally into a large number of shared-memory programs that use explicit synchronization to achieve their own higher-level consistency requirements. Even shared-memory models that provide sequential consistency often require such synchronization because they only support atomic operations on individual memory references. Implementation of complex memory operations involving multiple memory references

\(^3\)We sometimes refer to eventual consistency as periodic consistency to reflect the periodic transmission of updates.
requires additional synchronization [BZ91].

Through the use of application-aided consistency, applications can achieve the effect of consistency protocols such as weak consistency [DS86], release consistency [GLL+90, CBZ91], buffered consistency [SR92], or entry consistency [BZ91]. These consistency protocols are well documented and will not be described here. All build on the fact that updates need only be sent at synchronization points. They only differ in regards to the synchronization points at which updates must be transmitted and how much data must be transmitted.

Unify provides basic operations (see Section 5.3) that allow applications to specify when to update a shared segment and how to perform the update. By associating these update primitives with synchronization primitives, Unify applications can achieve the above consistency protocols. In addition, the flexibility to specify updates at arbitrary places in the code provides applications with the power to create their own consistency policies.

5.2 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 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 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 in-order consistent if all the log entries in the segment appear in the same relative order at all three sites. The replicas are 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 random access memory. Consequently, any consistency guarantee other than in-order consistency (e.g., out-of-order consistency) is of no value. Unify, however, supports two additional types of memory segments: associative memory and sequential memory. Given these two new types of memory, out-of-order consistency is quite meaningful and can substantially improve performance.

The relative ordering of values in an associative memory is uninteresting and of no use to
the application. Moreover, the relative ordering of the values in a sequential memory segment (e.g., a log file segment) often turn out to be irrelevant. For example, a shared log file might record timestamped error messages from several printer daemons. Because the data itself contains ordering information, there is no need to expend extra effort to insure that log entries appear in the correct chronological order across all cached copies.

In Unify, associative segments only require out-of-order consistency. Sequential segments, on the other hand, may require in-order or out-of-order consistency. Random segments are always in-order consistent.

We believe that the out-of-order consistency model provides a significant performance advantage for many distributed applications that might still require some form of temporal consistency.

5.3 Consistency Primitives

All workers that map a segment into their workspace see the same consistency properties for that segment.

Before describing the specific Unify routines that perform the above operations, we must define a few terms:

**Cache Set**: The cache set of a shared segment is the list of sites that currently have a cached copy of the segment.

**Copy Set**: The copy set of a shared segment is the list of sites that currently have a cached copy of the segment and want to be informed of any changes to the segment.

**Know_it_all Set**: The Know_it_all set of a shared segment is a list of sites chosen pseudo-randomly from the copy set. These sites are guaranteed to contain the most recent values at all times.

The difference between the Cache Set and the Copy Set may appear small at first, but the actions (and performance) of Unify depend heavily on the membership of each set [GYF92].

Assume an application wishes to use automatic temporal consistency. Unify takes responsibility for keeping the cached copies up to date. The operation to request this type of consistency is:

**Auto Updates(t, List_of_shared_segments)**

The application wants Unify to maintain consistency on a t time-unit granularity for all segments in List_of_shared_segments.
Using this operation, an application can achieve anything from sequential consistency \((t=0)\) to best-effort consistency \((t=\infty)\).

Application-aided temporal consistency requires application intervention to keep the cached copies at various sites consistent. Unify provides six routines to ensure the consistency of cached copies of a segment (see Table 5.3).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put Updates(List of shared segments)</td>
<td>transmits updates from segments in the list to every member of the copy set.</td>
</tr>
<tr>
<td>Get Updates(List of shared segments)</td>
<td>retrieve the most recent version of the shared segments from the last cache set member to invoke get_updates.</td>
</tr>
<tr>
<td>Lazy Put Updates(List of shared segments)</td>
<td>transmits the updates from all the segments in the list to every member of the know_it_all set.</td>
</tr>
<tr>
<td>Lazy Get Updates(List of shared segments)</td>
<td>retrieves the latest version of the shared segments in the list from a member from the know_it_all set (chosen at random).</td>
</tr>
<tr>
<td>Need Updates(List of shared segments)</td>
<td>allows a worker (really a node) to join the copy set of every segment specified in the list.</td>
</tr>
<tr>
<td>Dont Need Updates(List of shared segments)</td>
<td>removes a worker (really a node) from the copy set of every segment specified in the list.</td>
</tr>
</tbody>
</table>

Table 1: Six primitives to insure the consistency of cached copies of a segment

Depending on the desired level of weak consistency, an application (or a runtime library) will use Put Updates, Get Updates, or a combination of Lazy Put Updates and Lazy Get Updates. For instance, an application can achieve entry consistency (lazy updates) using only Get Updates. It can achieve release consistency (eager updates) using only Put Updates.

Lazy Put Updates and Lazy Get Updates allow applications to reduce the number of network messages and thereby increase scalability. An application might update a shared segment that will be used by a small subset of the copy set. To avoid the cost of transmitting the update to every node in the copy set, Unify selects a few members of the copy set to be members of the know_it_all set and only transmits the updates to those members. Later, when another site needs the latest copy of the shared segment, it contacts one of the members of the know_it_all set. A variety of algorithms can be used to decide the membership of the know_it_all set and enhance the scalability of consistency protocols. We expect the size of the know_it_all set to be relatively small, some fraction of the size of the copy set. In general, Lazy Put Updates and Lazy Get Updates allow Unify to scale up to a very large number of nodes, whereas Get Updates and Put Updates do not scale as well as the size of the copy set increases.
We considered using a hierarchical method (organizing nodes in a tree structure) to implement Lazy_{Put/Get}_Updates, but decided against it. If we used a hierarchical method [HF88, HFM88], each Lazy_Put_Update operation would traverse the tree towards the root (generating several messages). Similarly, each Lazy_Get_Update would propagate up through the tree, possibly to the root searching for the latest copy. Instead, we chose a single know_it_all set that restricts the upper bound on the amount of time it takes for either a put or a get operation. (The cost of get is a single round-trip message, whereas the cost of put is at most equal to the size of the know_it_all set or possibly less if the underlying network supports multicast.) We believe that our method will scale reasonably well assuming that the number of nodes remains less than 10,000 (implying a know_it_all size of about 100) and sharing domains contain on the order of 100 nodes interconnected via an efficient multicast network.

6 Synchronization

Synchronization in Unify sets it apart from all other DSM models. Most other models provide either locks, barriers, or both [GLL+90, CBZ91, BZ91, SR92], which severely limit performance and scalability. All these models 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.

However, many applications do not rely on ordered synchronization access, so Unify does not require (all) synchronization accesses to be sequentially consistent. Moreover, Unify does not insist that all sites see a synchronization access at the same time. Site A may see synchronization access S far earlier than site B, yet Unify does not block site A until site B sees the access. Instead, it allows site A to proceed immediately. This lack of sequentially consistent synchronization accesses results in better concurrency which ultimately translates into improved job throughput.

The key to supporting such 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.

Unify provides four eventcount operations.

Await_with_timeout(eventcount, count, timeout)

This call awaits for event to reach or surpass the value count. If the value of event has not reached the value count within timeout time units, then the call returns with a value indicating
that the call timed out.

**Advance(eventcount)**

This call uses unreliable multicast to inform all sites interested in *eventcount* that they should increment their value. The use of an unreliable delivery mechanism substantially reduces the number of messages generated by an advance operation and improves scalability.

**Event_sync(eventcount)**

This call brings the local value of *eventcount* into a consistent state. Since advance operations use an unreliable delivery mechanism, messages may be lost. If a site experiences a timeout from an *Await_with_timeout* call, it can invoke *Event_sync* to insure that its local value for *event* is up-to-date. This call may generate a large number of messages. However, in the expected case where messages are not lost, this routine will be called infrequently and only as a last resort.

Applications that use eventcounts should scale much better than applications that require sequentially consistent synchronization events. However, some applications depend on ordered synchronization events. The sequencer [RK79] abstraction and the associated *get_ticket* operation provide the necessary guarantees to insure that synchronization accesses are sequentially consistent when needed.

Unify provides two ticket operations.

**Get_ticket(sequencer)** *Get_ticket* returns the next number in sequence. Each number returned is guaranteed to be unique (that is, no two workers will get the same number) and densely sorted (no worker gets a given number unless all previous numbers have been given out. *Get_ticket* requires some form of distributed mutual exclusion.

**Get_R_ticket(R_sequencer)** *Get_R_ticket* returns an arbitrary unique number, but the set of numbers 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 sequence numbers only from its portion of the number space. Thus, *Get_R_ticket* avoids the distributed mutual exclusion necessary with the ordinary sequencer.

The *Get_R_ticket* routine is particularly useful for partitioning work among cooperating workers. Each worker needs to choose some arbitrary (but unique) portion of the problem to solve. In
Figure 3: An N x N matrix with M=9 workers is shown in the left part of the figure. Each worker computes values for its own section in each iteration. The section for each worker consists of nine segments. The figure on the right shows the sharing pattern for the Worker 5. The central segment (the white box) out of the nine segments is used only locally. The dark grey boxes represent shared segments that the worker must read from in each iteration. The light grey boxes represent shared segments that the worker updates and shares with its immediate neighbors.

In this case, we only must ensure that no two workers get the same portion of the problem, but can avoid the performance penalty of obtaining the next ticket in sequence.

7 An Example Application

To illustrate the scalability and performance advantages of the Unify consistency and synchronization primitives, we present a parallel implementation of the Jacobi algorithm to solve the Laplace heat equation and determine the temperature gradient on a square area given the temperature values at the area boundaries.

The square area is discretized into grid points according to some desired granularity. A matrix of values represents the temperature at each grid point. Given the temperature matrix, the Jacobi algorithm iteratively computes the temperature values at each grid point. During each iteration, each matrix element is updated by computing the average of the values of its four nearest neighbors.

Given a large matrix (N x N) and M workers, the matrix is subdivided into M sections. Each worker is responsible for computing the values in its section as shown in Figure 3. At the end of each iteration adjacent workers exchange computed values at section boundaries, introducing the need for a synchronization mechanism.

In the following, we first present a conventional implementation of the application using barriers for synchronization and then an implementation using Unify primitives.
7.1 Conventional Implementation

To avoid overwriting the value of an element before it is used (by one of the neighbors), workers use a “shadow” copy of the matrix to update values while leaving the “original” matrix unchanged. Each worker independently computes the new values for elements in its section of the “shadow” copy using the old values in the “original” matrix. The workers then synchronize by waiting on a barrier. After synchronizing, each worker switches its notion of the “original” and the “shadow” copy and repeats the algorithm until the solution converges.

This implementation does not allow maximum parallelism and does also not scale well for several reasons. First, conventional methods requires all workers to proceed in lock step. However, a worker only shares values with its immediate neighbors, and thus does not need to wait for other (non-adjacent) workers to compute their values before proceedings with the next iteration. Second, all the workers synchronize at a single barrier, which requires communication among all threads (unnecessary in this case) and poses a serious bottleneck as the multicomputer scales up.

7.2 Unify Implementation

In the Unify implementation, each section of the matrix assigned to a worker is further subdivided into nine segments as shown in Figure 3. The central portion of each section is not shared with any worker and can be treated as an ordinary segment with no overhead of sharing. The other eight (shared) segments are declared to be sequential segments (append-only) with application-aided temporal consistency and out-of-order spatial consistency. Each segment is written by one worker (say, worker 5 in Figure 3) and read by at most three workers. Workers that need adjacent values use the primitive \texttt{Need\_updates} to join the copy set of the adjacent segment.

In each iteration, a worker reads from each adjacent segment using the \texttt{read\_next\_item} primitive used with sequential segments (blocks the worker until the next item is available) and then writes (appends) new values to its shared segments. No explicit synchronization is needed, and a worker can get ahead of others as long as the values it needs are available. Moreover, updates to shared segments may appear out-of-order across the copies because each entry contains ordering information. Consequently, the operating system need not expend any effort to ensure in-order consistency.

This implementation clearly offers more parallelism (no single point of synchronization) and scales well as the updates to shared segments only propagate to a few sites. Other examples of
parallel applications that benefit from the Unify model have also been identified [GYF92].

8 Related Work

Unify has benefited considerably from previous research in the area of DSM models. Unify makes significant new contributions in some areas. In the following, we compare salient features of the Unify design with other DSM models.

8.1 Consistency

Many DSM proposals only support one consistency model. For example, IVY [Li86] and Mirage [FP89] only provide sequential consistency. Amber [JAL+89] also provides sequential consistency but migrates computations to the shared data, thus, serializing the concurrent accesses at a central place. Munin [CBZ91] and Midway [BZ91] provide one of the weak consistency models (entry consistency or release consistency) as well as sequential consistency. PVM [Sun91] provides sequential consistency, but the smallest update unit is the entire segment. Operating systems like Mach [YTR87, TRY+] do not provide DSM in the kernel, but allow applications to implement DSM outside the kernel.

One of the most important differences between Unify and other DSM models is that the others have not been designed with scalability as a central goal. Although the consistency model in Unify builds on previous work, Unify provides a new flexible sharing model by providing two classes of consistency protocols, namely, automatic and application-aided protocols. Under automatic methods, Unify introduces two new methods (eventual and best-effort consistencies) that are useful in a variety of distributed applications. Under application-aided models, Unify provides a range of alternatives instead of supporting a single weakly consistent model. In addition, Unify introduces spatial consistency and also supports different types of segments (random, sequential, associative) to exploit problem-specific semantics.

Clouds [RK88, DJA88] is an object-oriented DSM operating system that treats each object (or a logical segment) as a unit of consistency. The operating system and underlying hardware cooperate to maintain consistency of shared data. Clouds provides a general purpose synchronization mechanism that applications use to synchronize accesses to shared objects and Clouds exploits synchronization information in maintaining consistency by combining data transfers with

\footnote{Cleriton [Che86] first proposed the use of problem-specific semantics to reduce the cost of providing strict consistency and to enhance performance of distributed applications that share data.}
synchronization requests. This approach is similar to the release consistency protocol supported in Munin. A more recent version of Clouds [AAL92] provides a set of low-level consistency management primitives that allow an application (or a preprocessor) to implement a variety of consistency management protocols.

Although Clouds also supports multiple forms of consistency, it differs from Unify in several ways. First, application-specific consistency control in Clouds provides lower-level primitives than Unify and, thus, may suffer from application-induced errors. Second, the basic unit of sharing in Unify is a segment as opposed to a page in Clouds. Segments are logical units of protection and sharing that mirror the programming model better. Third, Unify does not associate an owner node with a shared object and thus is more scalable, compared to the Clouds approach which requires that all updates to go through the owner node. Fourth, Unify provides new models of automatic consistency that are not provided by Clouds. Finally, Unify provides scalable synchronization primitives.

Several large-scale multiprocessors [GLL+90, SR92, DSB86] exist that provide various forms of weak consistency. However, we have omitted discussion of those machines here due to lack of space and also because the work in Unify is aimed at building a loosely coupled, geographically distributed shared memory multicomputer.

8.2 Synchronization

Most DSM models provide some combination of locks, semaphores, spinlocks (useful on shared memory multiprocessors), monitors/condition variables, or barriers as the synchronization mechanism. Some of these mechanisms are easily implementable in hardware on multiprocessor machines. However, most of these mechanisms do not scale well in a distributed multicomputer spanning hundreds of sites. Scott and others [SMC91] describe algorithms to implement scalable synchronization primitives using spinlocks on shared memory machines. Unfortunately, spinning is an expensive operation in a loosely coupled multicomputer with an underlying unreliable communication facility.

Unify provides eventcounts and sequencers as synchronization primitives. The implementation of eventcounts does not require distributed mutual exclusion and thus scales well. In addition, eventcounts suffice as a synchronization mechanism for many applications that currently use barriers. Unify also provides a scalable version of sequencers (Get_R_Ticket) that avoids the bottleneck introduced by ordinary sequencers (typically implemented using a single coordinator or some form of distributed mutual exclusion).
Ivy also uses eventcounts as an underlying synchronization mechanism. Ivy is implemented using the Aegis (Apollo/Domain) operating system [LLD+83] and extends the eventcount mechanism in Aegis to work across a distributed environment. However, there are important differences between eventcounts in Ivy and Unify. Eventcounts in Ivy use a single owner (all operations are coordinated by a single node) which creates a potential bottleneck and increases congestion. Unify does not use a single node to keep track of the current value of an eventcount. Instead, eventcounts are implemented using a shared, sequential segment (without out-of-order spatial consistency) that accumulates all the advance operations. As opposed to Ivy, all await operations are local operations. This approach allows a high-level of parallelism among concurrent writers. In addition, Unify supports additional primitives (*await_with_timeout(), Event_sync(), and Get_R_ticket()) that combine to further reduce the bottlenecks and to increase scalability.

9 Current Status

We are currently implementing Unify in two different environments. First, we are implementing a library of routines that can be linked with Unify applications to create a distributed Unix application that emulates Unify and its sharing primitives. The distributed emulator experiences actual network latencies and errors. Second, we have begun implementation of a prototype native Unify operating system executing on bare hardware. We began by porting the VM XINU Operating System [Gri91] to the SPARC architecture to be used as the basis for Unify.

We are also investigating new methods for implementing both reliable and unreliable multicast messages. Multiple recipient messages dominate communication in a shared memory kernel and must be implemented efficiently. We continue to explore scalable algorithms for maintaining consistency and we intend to evaluate our current algorithms using the emulation library and eventually the prototype implementation.

9.1 Summary

We have described the design of the Unify multicomputer operating system and discussed how the consistency and synchronization mechanisms in Unify combine to offer scalable solutions to the problem of building a large scale, shared memory distributed computer. The principal ideas in Unify are to provide different types of segment types, a wide range of consistency models, and efficient synchronization primitives that together make it possible to realize a wide range of
problem-specific semantics. Furthermore, Unify provides applications knobs to carefully tune the performance depending on the scale and size of the parallelism desired. We are building a prototype implementation of Unify and expect to have interesting results in early Spring that will be included in the final version of this paper.

References


