

Multi-level Shared State for Distributed Systems *

DeQing Chen, Chunqiang Tang, Xiangchuan Chen,
Sandhya Dwarkadas, and Michael L. Scott

Computer Science Department, University of Rochester
{lukechen,sarmor,chenxc,sandhya,scott}@cs.rochester.edu

Abstract

As a result of advances in processor and network speeds, more and more applications can productively be spread across geographically distributed machines. In this paper we present a transparent system for memory sharing, *InterWeave*, developed with such applications in mind. *InterWeave* can accommodate hardware coherence and consistency within multiprocessors (level-1 sharing), software distributed shared memory (S-DSM) within tightly coupled clusters (level-2 sharing), and version-based coherence and consistency across the Internet (level-3 sharing). *InterWeave* allows processes written in multiple languages, running on heterogeneous machines, to share arbitrary typed data structures as if they resided in local memory. Application-specific knowledge of minimal coherence requirements is used to minimize communication. Consistency information is maintained in a manner that allows scaling to large amounts of shared data. In C, operations on shared data, including pointers, take precisely the same form as operations on non-shared data. We demonstrate the ease of use and efficiency of the system through an evaluation of several applications. In particular, we demonstrate that *InterWeave*'s support for sharing at higher (more distributed) levels does not reduce the performance of sharing at lower (more tightly coupled) levels.

1 Introduction

Advances in processing speed and network bandwidth are creating new interest in such ambitious distributed applications as interactive data mining, remote scientific visualization, computer-supported collaborative work, and intelligent environments. These applications are characterized both by the need for high-end parallel computing and by the need to coordinate widely distributed users, devices, and data repositories. Increasingly, the parallel computing part can make productive use of the parallelism afforded by comparatively inexpensive and widely available clusters of

*This work was supported in part by NSF grants CCR-9702466, CCR-9705594, EIA-9972881, CCR-9988361, EIA-0080124, and CCR-0204344.

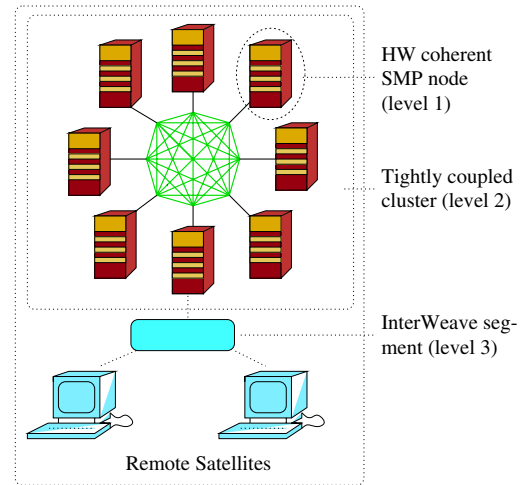


Figure 1. *InterWeave*'s target environment.

symmetric multiprocessors (SMPs). The more distributed components may need to span the Internet.

Conceptually, many of these applications seem easiest to describe in terms of some sort of *shared state*. Many programmers—particularly those who are connecting together components developed for small and mid-size multiprocessors—would like to capture shared state with a shared-memory programming model. In order to meet this demand, we are developing a system, known as *InterWeave* [8], that allows the programmer to map shared data into program components regardless of location or machine type, and to transparently access that data once mapped.

InterWeave represents a merger and extension of our previous *Cashmere* [19, 20] and *InterAct* [15] projects. Once shared data has been mapped, *InterWeave* can support hardware coherence and consistency within multiprocessors (*level-1* sharing), *Cashmere*-style software distributed shared memory (S-DSM) within tightly coupled clusters (*level-2* sharing), and *InterAct*-style version-based coherence and consistency across the Internet (*level-3* sharing). Figure 1 provides a pictorial representation of the target environment.

InterWeave has been designed to maximize the lever-

age of available hardware support, and to minimize the extent to which sharing at the higher (more distributed) levels might impact the performance of sharing at the lower (more tightly coupled) levels. At levels 1 and 2, InterWeave inherits Cashmere’s integration of intra-SMP hardware cache coherence with cluster-level VM-based lazy release consistency. In particular, it employs *two-way diffing* to avoid the need for TLB shutdown when processes synchronize across nodes [19], and relies on low-latency user-level messages for efficient synchronization, directory management, and write-notice propagation [20]. In a similar vein, consistency at level 3 employs the twins, diffs, write notices, and home-node copies already maintained at level 2.

At the third level, data in InterWeave evolves through a series of consistent versions. Application-specific knowledge of minimal coherence requirements is used to minimize communication. When beginning a read-only critical section on a logical grouping of data (a *segment*), InterWeave uses a programmer-specified predicate to determine whether the currently cached version, if any, is “recent enough” to use. Several coherence models (notions of “recent enough”) are built into the InterWeave system; others can be defined by application programmers. When the application desires consistency across segment boundaries, to avoid causality loops, we invalidate mutually-inconsistent versions using a novel hashing mechanism that captures the history of a segment in a bounded amount of space. S-DSM-like twins and diffs allow us to update stale segments economically.

In keeping with wide-area distribution, InterWeave allows processes at level 3 to be written in multiple languages and to run on heterogeneous machine architectures, while sharing arbitrary typed data structures as if they resided in local memory [21]. In C, operations on shared data, including pointers, take precisely the same form as operations on non-shared data. Like CORBA and many older RPC systems, InterWeave employs a type system based on a machine- and language-independent interface description language (IDL).¹ When transmitting data between machines, we convert between the local data format (as determined by language and machine architecture) and a standard *InterWeave wire format*. We also *swizzle* pointers [23] so that they can be represented locally using ordinary machine addresses.

Recognizing that the performance tradeoffs between function shipping and data migration/caching are application-dependent, we have designed InterWeave to complement existing RPC and RMI systems. Programmers can choose on a function-by-function basis whether to access data directly or to invoke an operation on a machine

¹InterWeave’s IDL is currently based on Sun XDR, but this is not an essential design choice. InterWeave could easily be modified to work with other IDLs.

at which the data is believed to reside. When choosing the latter option, the presence of the InterWeave library allows a program to use genuine reference parameters as an alternative to deep-copy value parameters.

We describe the design of InterWeave in more detail in Section 2. We then describe our implementation in Section 3, with an emphasis on the coherence, consistency, and communication mechanisms. Performance results for applications in iterative, interactive data mining; remote scientific visualization; and multi-user collaboration appear in Section 4. We compare our design to related work in Section 5 and conclude with a discussion of status and plans in Section 6.

2 InterWeave Design

The InterWeave programming model assumes a distributed collection of servers and clients. Servers maintain persistent copies of shared data, and coordinate sharing among clients. Clients in turn must be linked with a special InterWeave library, which arranges to map a cached copy of needed data into local memory, and to update that copy when appropriate.

2.1 Data Allocation

The unit of sharing in InterWeave is a self-descriptive data *segment* (a heap) within which programs allocate strongly typed *blocks* of memory.² Every segment is specified by an Internet URL. The blocks within a segment are numbered and optionally named. By concatenating the segment URL with a block name or number and optional offset (delimited by pound signs), we obtain a *machine-independent pointer (MIP)*: “foo.org/path#block#offset”. To accommodate heterogeneous data formats, offsets are measured in primitive data units—characters, integers, floats, etc.—rather than in bytes.

Every segment is managed by an InterWeave server at the IP address corresponding to the segment’s URL. Different segments may be managed by different servers. Assuming appropriate access rights, the `IW_open_segment()` library call communicates with the appropriate server to open an existing segment or to create a new one if the segment does not yet exist.³ The call returns an opaque *han-*

²Like distributed file systems and databases, and unlike systems such as PerDiS [11], InterWeave requires manual deletion of data; there is no automatic garbage collection. A web-based perusal tool, comparable to a file-system browser, will allow a user or system administrator to search for orphaned data.

³Authentication and access control in InterWeave are currently based on a simple public key mechanism. Access keys can be specified at segment creation time or changed later by any client that successfully acquires write access.

dle that can be passed as the initial argument in calls to `IW_malloc()`:

```
IW_handle_t h = IW_open_segment(url);
IW_wl_acquire(h);          /* write lock */
my_type* p = (my_type*)
    IW_malloc(h, my_type_desc);
*p = ...
IW_wl_release(h);
```

As in multi-language RPC systems, the types of shared data in InterWeave must be declared in IDL. The InterWeave IDL compiler translates these declarations into the appropriate programming language(s) (C, C++, Java, Fortran). It also creates initialized *type descriptors* that specify the layout of the types on the specified machine. The descriptors must be registered with the InterWeave library prior to being used, and are passed as the second argument in calls to `IW_malloc()`. These conventions allow the library to translate to and from wire format, ensuring that each type will have the appropriate machine-specific byte order, alignment, etc. in locally cached copies of segments.

Level-3 synchronization takes the form of reader-writer locks. A process must hold a writer lock on a segment in order to allocate, free, or modify blocks. The lock routines take a segment handle as parameter. Within a tightly coupled cluster or a hardware-coherent node, a segment that is locked at level 3 may be shared using data-race-free [1] memory semantics.

Given a pointer to a block in an InterWeave segment, or to data within such a block, a process can create a corresponding MIP:

```
IW_mip_t m = IW_ptr_to_mip(p);
```

This MIP can then be passed to another process through a message, a file, or an argument of a remote procedure in RPC-style systems. Given appropriate access rights, the other process can convert back to a machine-specific pointer:

```
my_type *p = (my_type*) IW_mip_to_ptr(m);
```

The `IW_mip_to_ptr` call reserves space for the specified segment if it is not already locally cached (communicating with the server if necessary to obtain layout information for the specified block), and returns a local machine address. Actual data for the segment will not be copied into the local machine until the segment is locked.

It should be emphasized that `IW_mip_to_ptr()` is primarily a bootstrapping mechanism. Once a process has one pointer into a data structure, any data reachable from that pointer can be directly accessed in the same way as local data, even if embedded pointers refer to data in other segments. InterWeave's pointer-swizzling and data-conversion

mechanisms ensure that such pointers will be valid local machine addresses. It remains the programmer's responsibility to ensure that segments are accessed only under the protection of reader-writer locks.

2.2 Coherence

InterWeave's goal is to support seamless sharing of data using ordinary reads and writes, regardless of location. Unfortunately, given the comparatively high and variable latencies of even local-area networks, traditional hardware-inspired coherence and consistency models are unlikely to admit good performance in a distributed environment. Even the most relaxed of these models guarantees a consistent view of *all* shared data among *all* processes at synchronization points, resulting in significant amounts of communication. To reduce this overhead, InterWeave exploits the fact that processes in a distributed application can often accept a significantly more relaxed—and hence less communication-intensive—notation of coherence. Depending on the application, it may suffice to update a cached copy of a segment at regular (temporal) intervals, or whenever the contents have changed “enough to make a difference,” rather than after every change. When updating data, we require that a process have exclusive write access to the most recent version of the segment. When reading, however, we require only that the currently cached version be “recent enough” to satisfy the needs of the application.

InterWeave currently supports six different definitions of “recent enough”. It is also designed in such a way that additional definitions (coherence models) can be added easily. Among the current models, *Full* coherence always obtains the most recent version of the segment; *Strict* coherence obtains the most recent version *and* excludes any concurrent writer; *Null* coherence always accepts the currently cached version, if any (the process must explicitly override the model on an individual lock acquire in order to obtain an update); *Delta* coherence [17] guarantees that the segment is no more than x versions out-of-date; *Temporal* coherence guarantees that it is no more than x time units out of date; and *Diff-based* coherence guarantees that no more than $x\%$ of the primitive data elements in the segment are out of date. In all cases, x can be specified dynamically by the process. All coherence models other than *Strict* allow a process to hold a read lock on a segment even when a writer is in the process of creating a new version.

When a process first locks a shared segment, the InterWeave library obtains a copy from the segment's server. At each subsequent read-lock acquisition, the library checks to see whether the local copy of the segment is “recent enough”. If not, it obtains a version update from the server. An adaptive polling/notification protocol, described in Section 3.3, often allows the implementation to avoid communication with the server when updates are not required.

Twin and diff operations [6], extended to accommodate heterogeneous data formats, allow the implementation to perform an update in time proportional to the fraction of the data that has changed.

Unless otherwise specified, lock acquisitions default to Full coherence. The creator of a segment can specify an alternative coherence model if desired, to be used by default whenever any process locks that particular segment. An individual process may also establish its own default for a given segment, and may override this default for individual critical sections. Different processes (and different fragments of code within a given process) may therefore use different coherence models for the same segment. These models are entirely compatible: the server for a segment always has the most recent version; the model used by a given process at a given time simply determines how it decides if its own cached copy is recent enough.

The server for a segment need only maintain a copy of the segment’s most recent version. The API specifies that the current version of a segment is always acceptable as an update to a client, and since processes cache whole segments, they never need an “extra piece” of an old version. To minimize the cost of segment updates, the server maintains a timestamp on each block of each segment, so that it can avoid transmitting copies of blocks that have not changed. As partial protection against server failure, InterWeave periodically checkpoints segments and their metadata to persistent storage. The implementation of real fault tolerance is a subject of future work.

As noted in Section 1, an SDSM-style “level-2” sharing system such as Cashmere can play the role of a single node at level 3. Any process in a level-2 system that obtains a level-3 lock does so on behalf of its entire level-2 system, and may share access to the segment with its level-2 peers. If level-3 lock operations occur in more than one level-2 process, the processes must coordinate their activities (using ordinary level-2 synchronization) so that operations are seen by the server in an appropriate order. Working together, Cashmere and InterWeave guarantee that updates are propagated consistently, and that protocol overhead required to maintain coherence is not replicated at levels 2 and 3. Further details appear in Section 3.

2.3 Consistency

Without additional mechanisms, in the face of multi-version relaxed coherence, the versions of segments currently visible to a process might not be mutually consistent. Specifically, let A_j refer to version j of segment A . If B_k was created using information found in A_j , then previous versions of A are causally incompatible with B_k ; a process that wants to use B_k (and that wants to respect causality) should invalidate any cached segment version A_i , $i < j$.

To support this invalidation process, we would ideally

like to tag each segment version, automatically, with the names of all segment versions on which it depends. Then whenever a process acquired a lock on a segment the library would check to see whether that segment depends on newer versions of any other segments currently locally cached. If so, the library would invalidate those segments. The problem with this scheme, of course, is that the number of segments in the system—and hence the size of tags—is unbounded. In Section 3.2 we describe a mechanism based on hashing that achieves the same effect in bounded space, at modest additional cost.

To support operations on groups of segments, we allow their locks to be acquired and released together. Locks that are acquired together are acquired in a predefined total order to avoid deadlock. Write locks released together make each new segment version appear to be in the logical past of the other, ensuring that a process that acquires the locks together will never obtain the new version of one without the other. To enhance the performance of the most relaxed applications, we allow an individual process to “opt out” of causality on a segment-by-segment basis. For sharing levels 1 and 2 (hardware coherence within SMPs, and software DSM within clusters), consistency is guaranteed for data-race-free programs.

3 Implementation

The underlying implementation of InterWeave can be divided into four relatively independent modules:

- the memory management module, which provides address-independent storage for segments and their associated metadata;
- the modification detection module, which creates wire-format diffs designed to accommodate heterogeneity and minimize communication bandwidth;
- the coherence and consistency module, which obtains updates from the server when the cached copy of a segment is no longer recent enough, or is inconsistent with the local copies of other segments; and
- the communication module, which handles efficient communication of data between servers and clients.

The memory management and modification detection modules are described in detail in a companion paper [21]. We describe them briefly in the first subsection below, and then focus in the remaining subsections on the coherence/consistency and communication modules.

3.1 Memory Management and Modification Detection

As described in Section 2, InterWeave presents the programmer with two granularities of shared data: *segments*

and *blocks*. Each block must have a well-defined type, but this type can be a recursively defined structure of arbitrary complexity, so blocks can be of arbitrary size. Every block has a serial number within its segment, assigned by `IW_malloc()`. It may also have a symbolic name, specified as an additional parameter. There is no a priori limit on the number of blocks in a segment, and blocks within the same segment can be of different types and sizes.

When a process acquires a write lock on a given segment, the InterWeave client library asks the operating system to write protect the pages that comprise the local copy of the segment. When a page fault occurs, the `SIGSEGV` signal handler, installed by the library at program startup time, creates a pristine copy, or *twin* [6], of the page in which the write fault occurred. It saves a pointer to that twin for future reference, and then asks the operating system to re-enable write access to the page.

When a process releases a write lock, the library performs a word-by-word diff of modified pages and their twins. It then converts this diff to a machine-independent wire format that expresses changes in terms of segments, blocks, and primitive data unit offsets, rather than pages and bytes, and that compensates for byte order, word size, and alignment. When a client acquires a lock and determines that its copy of the segment is not recent enough, the server builds a similar diff that describes the data that have changed between the client’s outdated copy and the master copy at the server.

Both translations between local and wire format—for updates to the server at write lock release and for updates to the client at lock acquisition—are driven by type descriptors, generated by the InterWeave IDL compiler, and provided to the InterWeave library via the second argument to `IW_malloc()` calls. The content of each descriptor specifies the substructure and machine-specific layout of its type.

To accommodate reference types, InterWeave relies on pointer swizzling [23]. Briefly, swizzling uses type descriptors to find all (machine-independent) pointers within a newly-cached or updated segment, and converts them to pointers that work on the local machine. Pointers to segments that are not (yet) locally cached point into reserved but unmapped pages where data will lie once properly locked. The set of segments currently cached on a given machine thus displays an “expanding frontier” reminiscent of lazy dynamic linking.

3.2 Coherence and Consistency

Each server maintains an up-to-date copy of each of the segments for which it is responsible, and controls access to those segments. For each segment, the InterWeave server keeps track of blocks and *subblocks*. Each subblock comprises a small contiguous group of primitive data elements from the same block. For each modest-sized block in each

segment, and for each subblock of a larger block, the server remembers the version number of the segment in which the content of the block or subblock was most recently modified. This convention strikes a compromise between the size of server-to-client diffs and the size of server-maintained metadata.

At the time of a lock acquire, a client must decide whether its local copy of the segment needs to be updated. (This decision may or may not require communication with the server; see Section 3.3.) If an update is required, the client sends the server the (out-of-date) version number of the local copy. The server then identifies the blocks and subblocks that have changed since the last update to this client, constructs a wire-format diff, and returns it to the client.

Hash-Based Consistency. To ensure inter-segment consistency, we use a simple hash function to compress the dependence history of segments. Specifically, we tag each segment version S_i with an n -slot vector timestamp, and choose a global hash function h that maps segment identifiers into the range $[0..n - 1]$. Slot j in the vector indicates the maximum, over all segments P whose identifiers hash to j , of the most recent version of P on which S_i depends. When acquiring a lock on S_i , a process checks each of its cached segment versions Q_k to see whether k is less than the value in slot $h(Q)$ of S_i ’s vector timestamp. If so, the process invalidates Q_k . Hash collisions may result in unnecessary invalidations, but these affect performance only, not correctness.

To support the creation of segment timestamps, each client maintains a local master timestamp. When the client acquires a lock on any segment (read or write) that forces it to obtain a new version of a segment from a server, the library updates the master timestamp with any newer values found in corresponding slots of the timestamp on the newly obtained segment version. When releasing a write lock (thereby creating a new segment version), the process increments the version number of the segment itself, updates its local timestamp to reflect that number, and attaches this new timestamp to the newly-created segment version.

Integration with 2-Level System. When a tightly coupled cluster, such as a Cashmere-2L system, uses an InterWeave segment, the cluster appears as a single client to the segment server. The client’s local copy of the segment is kept in cluster-wide shared memory.

Figure 2 pictorially represents a sequence of actions performed by a level-2 system. (Details on our level-2 coherence protocol can be found in previous work [19].) The timelines in the figure flow from left to right, and represent three processors within a tightly coupled cluster. In the current implementation, we designate a single node within the cluster to be the segment’s *manager* node (in this case processor P0). All interactions between the level-2 system and

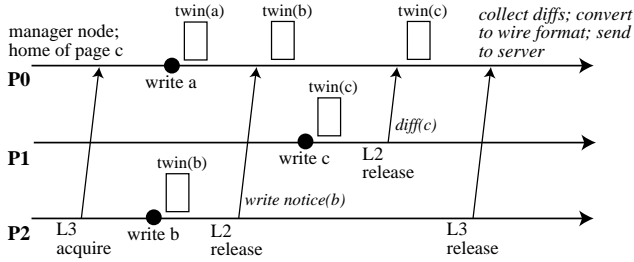


Figure 2. Coherence actions at levels 2 and 3.

the segment's InterWeave server go through the manager node. During the period between a level-3 (InterWeave) write lock acquire and release, the manager node ensures that modifications made within the level-2 system can be identified through the use of twins and diffs.

InterWeave achieves its goal of minimizing additional coherence actions by piggybacking as far as possible on existing level-2 operations. Three different scenarios are illustrated in the figure. First, as illustrated on the P0 timeline, the manager node creates a twin for a page if it experiences a write fault. If the manager is not the level-2 home node for the page, then this twin is used for both level-2 and level-3 modification detection purposes. If the manager node is the level-2 home node, then this twin is needed for level 3 only. Second, as illustrated by page *b*, the manager creates a level-3 twin if it receives a write notice from another node in the cluster (P2) and must invalidate the page. Third, as illustrated by page *c*, the manager creates a twin for level-3 purposes (only) if it receives a level-2 diff from another node in the cluster (P1).

On a level-3 release, the manager node compares any level-3 twins to the current content of the corresponding pages in order to create diffs for the InterWeave server. Overhead is thus incurred only for those pages that are modified and, in practice, the number of additional twins created is fairly low.

3.3 Communication

In our current implementation each InterWeave server takes the form of a daemon process listening on a well-known port at a well-known Internet address for connection requests from clients. The server keeps metadata for each active client of each segment it manages, as well as a master copy of the segment's data.

Each InterWeave client maintains a pair of TCP connections to each server for which it has locally cached copies of segments. One connection is used for client requests and server responses. The other is used for server notifications. Separation of these two categories of communication allows them to be handled independently. All communication between clients and servers is aggregated so as to minimize

the number of messages exchanged (and thereby avoid extra per-message overhead).

Servers use a heartbeat mechanism to identify dead clients. If a client dies while holding a write lock or a read lock with Strict coherence, the server reverts to the previous version of the segment. If the client was not really dead (its heartbeat was simply delayed), its subsequent release will fail.

Several protocol optimizations minimize communication between clients and servers in important common cases. (Additional optimizations, not described here, minimize the cost of modification detection and conversion to and from wire format [21].) First, when only one client has a copy of a given segment, the client will enter *exclusive* mode, allowing it to acquire and release locks (both read and write) an arbitrary number of times, with no communication with the server whatsoever. This optimization is particularly important for high-performance clients such as Cashmere clusters. If other clients appear, the server sends a message requesting a summary diff, and the client leaves exclusive mode.

Second, a client that finds that its local copy of a segment is usually recent enough will enter a mode in which it stops asking the server for updates. Specifically, every locally cached segment begins in *polling* mode: the client will check with the server on every read lock acquire to see if it needs an update (temporal coherence provides an exception to this rule: no poll is needed if the window has yet to close). If three successive polls fail to uncover the need for an update, the client and server will switch to *notification* mode. Now it is the server's responsibility to inform the client when an update is required (it need only inform it once, not after every new version is created). If three successive lock acquisition operations find notifications already waiting, the client and server will revert to polling mode.

Third, the server maintains a cache of diffs that it has received recently from clients, or collected recently itself, in response to client requests. These cached diffs can often be used to respond to future requests, avoiding redundant collection overhead.

Finally, as in the TreadMarks SDSM system [4], a client that repeatedly modifies most of the data in a segment will switch to a mode in which it simply transmits the whole segment to the server at every write lock release. This *no-diff* mode eliminates the overhead of *mprotects*, page faults, and the creation of twins and diffs.

4 Performance Results

InterWeave currently runs on Alpha, Sparc, x86, and MIPS processors, under Windows NT, Linux, Solaris, Tru64 Unix, and IRIX. Together, the server and client library comprise approximately 31,000 lines of heavily com-

mented C++ code. Our uniprocessor results were collected on Sun Ultra 5 workstations with 400 MHz Sparc v9 processors and 128 MB of memory, running SunOS 5.7, and on 333 MHz Celeron PCs with 256 MB of memory, running Linux 6.2. Our Cashmere cluster is a collection of AlphaServer 4100 5/600 nodes, each with four 600 MHz 21164A processors, an 8 MB direct-mapped board-level cache with a 64-byte line size, and 2 GBytes of memory, running Tru64 Unix 4.0F. The nodes are connected by a Memory Channel 2 system area network, which is used for tightly-coupled sharing. Connection to the local area network is via TCP/IP over 100Mb Ethernet.

4.1 Coherence Model Evaluation

We use a data mining application [16] to demonstrate the impact of InterWeave’s relaxed coherence models on network bandwidth and synchronization latency. Specifically, the application performs incremental sequence mining on a remotely located database of *transactions* (e.g. retail purchases). Each transaction in the database (not to be confused with transactions *on* the database) comprises a set of *items*, such as goods that were purchased together. Transactions are ordered with respect to each other in time. The goal is to find sequences of items that are commonly purchased by a single customer in order over time.

In our experimental setup, the database server (itself an InterWeave client) reads from an active database whose content continues to grow. As updates arrive the server incrementally maintains a summary data structure (a lattice of item sequences) that is used by mining queries. Each node in the lattice represents a sequence that has been found with a frequency above a specified threshold. The lattice is represented by a single InterWeave segment; each node is a block in that segment. Each data mining client, representing a distributed, interactive interface to the mining system, is also an InterWeave client. It executes a loop containing a reader critical section in which it performs a simple query.

Our sample database is generated by tools from IBM research [18]. It includes 100,000 customers and 1000 different items, with an average of 1.25 transactions per customers and a total of 5000 item sequence patterns of average length 4. The database size is 20MB.

The summary structure is initially generated using full the database. The server then repeatedly updates the structure using an additional 1% of the database each time. Because the summary structure is large, and changes slowly over time, it makes sense for each client to keep a local cached copy of the structure and to update only the modified data as the database evolves. Moreover, since the data in the summary are statistical in nature, their *values* change slowly over time, and clients do not need to see each incremental change. Delta or diff coherence will suffice, and can dramatically reduce communication overhead. To illustrate

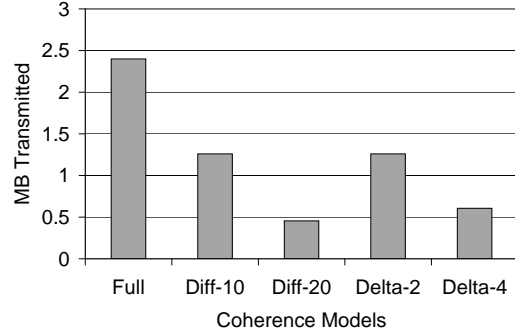


Figure 3. Sequence mining: bandwidth required under different coherence models.

these effects, we measure the network bandwidth required by each client for summary data structure updates as the database grows and the database server finds additional sequences.

Figure 3 shows the total bandwidth requirement as the client relaxes its coherence model. The leftmost bar represents the bandwidth requirement if the client uses the Full coherence model (Section 2.2). The other four bars show the bandwidth requirements if the client uses Diff and Delta coherence with different thresholds. Using Diff coherence with a threshold of 20% (i.e., consider a cached copy to be “recent enough” if no more than 20% of its primitive data elements are out of date), we see a savings of almost 75%.

4.2 3-Level System for Parallel Applications

To illustrate the interaction between InterWeave shared state, managed across the Internet, and software distributed shared memory, running on a tightly coupled cluster, we collected performance measurements for remote visualization and steering of two pre-existing scientific simulations: the Splash-2 Barnes-Hut N-body benchmark, and a CFD stellar dynamics application known as Astroflow [10]. Barnes-Hut is written in C. Astroflow is written in Fortran. Both simulations run on four nodes of our AlphaServer cluster. Barnes-Hut repeatedly computes new positions for 16,384 bodies. Astroflow computes on a 256×256 discretized grid. In both cases, the progress of the simulation can be observed and modified using a visualization and steering “satellite” that runs on a remote workstation. The Astroflow satellite is a pre-existing Java program, originally designed to read from a checkpoint file, but modified for our purposes to share data with the simulator via InterWeave. The Barnes-Hut satellite was written from scratch (in C) for this experiment.

In both applications, the simulator uses a write lock to update the segment that it shares with the satellite. The Barnes-Hut satellite uses a relaxed read lock with Temporal coherence to obtain an effective frame rate of 15 frames

per second. In Astroflow the simulation proceeds slowly enough that Full coherence requires negligible bandwidth.

To assess the baseline overhead of InterWeave we linked both simulators with the InterWeave library, but ran them without connecting to a satellite. Though the cluster must communicate with the InterWeave server to create its initial copy of the simulation data, the *exclusive mode* optimization (Section 3.3) eliminates the need for further interaction, and the overall impact on performance is negligible.

To assess the overhead of InterWeave in the presence of a satellite, we constructed, by hand, versions of the simulators that use explicit messaging over TCP/IP to communicate with the satellite (directly, without a server). We then ran these versions on the standard Cashmere system, and compared their performance to that of Cashmere working with InterWeave. Results for Barnes-Hut appear in Figure 4. (For Astroflow, both the messaging and InterWeave versions have such low communication rates that the impact on performance is negligible.) In all cases the satellite was running on another Alpha node, communicating with the cluster and server, if any, via TCP/IP over 100Mb Ethernet. Each bar gives aggregate wall-clock time for ten iteration steps. The labels on pairs of bars indicate the number of nodes and the total number of processors involved in each experiment. In the first three pairs a single processor was active in each node. In the final pair, four processors per node were active. The “C” bars are for explicit messaging code running on standard Cashmere; the “IW” bars are for Cashmere working with InterWeave. The C bars are subdivided to show the overhead of communication; the IW bars also show the (comparatively small) overhead of data translation and the coherence protocol. For this particular sharing scenario, much of the shared data is modified in every interval. InterWeave therefore switches, automatically, to *no-diff* mode to minimize the cost of tracking modifications.

4.3 API Ease-of-Use

The changes required to adapt Barnes-Hut and Astroflow to work with InterWeave were small and isolated. No special code is required to control the frequency of updates (one can adjust this in the satellite simply by specifying a temporal bound on relaxed coherence). No assumptions need to be embedded regarding the number of satellites (one can launch an arbitrary number of them, on multiple workstations, and each will connect to the server and monitor the simulation). No knowledge of networking or connection details is required, beyond the character-string name of the segment shared between the simulator and the satellite. While the matter is clearly subjective, we find the InterWeave code to be significantly simpler, easier to understand, and faster to write than the message-passing version.

In a separate experiment, we used InterWeave to de-

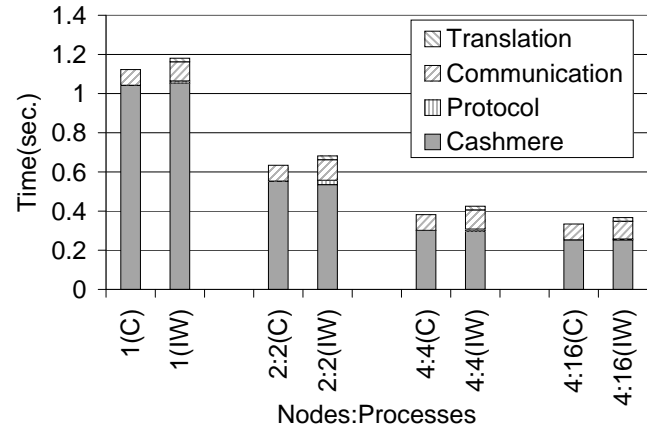


Figure 4. Overhead of InterWeave library and communication during Barnes-Hut remote visualization.

velop, from scratch, a distributed calendar program. The program was originally written with about two weeks of part-time effort by a first-year graduate student. Subsequent minor modifications served primarily to cope with changes in the API as InterWeave evolved.

The program maintains appointment calendars for a dynamically changing group of individuals. Users can create or delete a personal calendar; view appointments in a personal calendar or, with permission, the calendars of others; create or delete individual appointments; propose a group meeting, to be placed in the calendars of a specified group of users; or accept or reject a meeting proposal.

A single global segment, accessed by all clients, contains a directory of users. For each user, there is an additional segment that contains the user’s calendar. Within each user calendar there is a named block for each day on which appointments (firm or proposed) exist. The name of the block is a character string date. To obtain a pointer to Jane Doe’s calendar for April 1, we say `IW_mip_to_ptr("iw.somewhere.edu/cal/jane#04-01-2001")`.

The calendar program comprises 1250 lines of C++ source, approximately 570 of which are devoted to a simple command-line user interface. There are 68 calls to InterWeave library routines, spread among about a dozen user-level functions. These calls include 3 reader and 10 writer lock acquire/release pairs, 17 additional lock releases in error-checking code, and a dozen `IW_mip_to_ptr` calls that return references to segments.

In comparison to messaging code, the InterWeave calendar program has no message buffers, no marshaling and unmarshaling of parameters, and no management of TCP connections. (These are all present in the InterWeave library, of course, but the library is entirely general, and can be reused

by other programs.) Instead of an application-specific protocol for client-server interactions, the InterWeave code has reader-writer locks, which programmers, in our experience, find significantly more straightforward and intuitive.

5 Related Work

InterWeave finds context in an enormous body of related work—far too much to document in this paper. We focus here on some of the most relevant systems in the literature; additional discussion can be found in the TR version of this paper [9].

Dozens of object-based systems attempt to provide a uniform programming model for distributed applications. Many are language specific; many of the more recent of these are based on Java. Language-independent distributed object systems include PerDiS [11], Legion [13], Globe [22], Microsoft’s DCOM, and various CORBA-compliant systems. Globe replicates objects for availability and fault tolerance. PerDiS and a few CORBA systems (e.g. Fresco [14]) cache objects for locality of reference. Unfortunately, object-oriented update propagation, typically supported either by invalidate and resend on access or by RMI-style mechanisms, tends to be inefficient (re-sending a large object or a log of operations). Equally significant from our point of view, there are important applications (e.g. compute intensive parallel applications) that do not employ an object-oriented programming style.

At least two early S-DSM systems provided support for heterogeneous machine types. Toronto’s Mermaid system [25] allowed data to be shared across more than one type of machine, but only among processes created as part of a single run-to-completion parallel program. All data in the same VM page was required to have the same type, and only one memory model—sequential consistency—was supported. CMU’s Agora system [5] supported sharing among more loosely-coupled processes, but in a significantly more restricted fashion than in InterWeave. Pointers and recursive types were not supported, all shared data had to be accessed indirectly through a local mapping table, and only a single memory model (similar to processor consistency) was supported.

Friedman [12] and Agrawal et al. [2] have shown how to combine certain pairs of consistency models in a non-version-based system. Alonso et al. [3] present a general system for relaxed, user-controlled coherence. Khazana [7] also proposes the use of multiple consistency models. The TACT system of Yu et al. [24] allows coherence and consistency requirements to vary continuously in three orthogonal dimensions. Several of InterWeave’s built-in coherence models are similarly continuous, but because our goal is to reduce read bandwidth and latency, rather than to increase availability (concurrency) for writes, we insist on strong se-

manatics for writer locks. To the best of our knowledge, InterWeave is the first system to provide a general framework in which the user can define application-specific coherence models.

6 Conclusions and Future Work

We have described a run-time system, InterWeave, that allows processes to access shared data transparently using ordinary reads and writes. InterWeave is, to the best of our knowledge, the first such system to seamlessly and efficiently span the spectrum from hardware cache coherence within SMP nodes, through software distributed shared memory on tightly-coupled clusters, to relaxed, version-based coherence across the Internet. It is also, we believe, the first to fully support shared memory across heterogeneous machine types and languages.

We have demonstrated the efficiency and ease of use of the system through an evaluation on both real applications and artificial benchmarks. Experience to date indicates that users find the API conceptually appealing, and that it allows them to build new programs significantly more easily than they can with RPC or other message passing paradigms. For applications in which RPC-style function shipping is required for good performance, InterWeave provides enhanced functionality via genuine reference parameters.

Quantitative measurements indicate that InterWeave is able to provide sharing in a distributed environment with minimal impact on the performance of more tightly-coupled sharing. InterWeave facilitates the use of relaxed coherence and consistency models that take advantage of application-specific knowledge to greatly reduce communication costs, and that are much more difficult to implement in hand-written message-passing code. We are actively collaborating with colleagues in our own and other departments to employ InterWeave in three principal application domains: remote visualization and steering of high-end simulations (enhancing the Astroflow visualization described in Section 4.2), incremental interactive data mining (Section 4.1), and human-computer collaboration in richly instrumented physical environments.

Acknowledgments

Srinivasan Parthasarathy developed the InterAct system, and participated in many of the early design discussions for InterWeave. Eduardo Pinheiro wrote an earlier version of InterWeave’s heterogeneity management code. We are grateful to Amy Murphy and Chen Ding for their insightful suggestions on the content of this paper.

References

- [1] S. V. Adve and M. D. Hill. A Unified Formulation of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.
- [2] D. Agrawal, M. Choy, H. V. Leong, and A. K. Singh. Mixed Consistency: A Model for Parallel Programming. In *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994.
- [3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Trans. on Database Systems*, 15(3):359–384, Sept. 1990.
- [4] C. Amza, A. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd Intl. Symp. on High Performance Computer Architecture*, San Antonio, TX, Feb. 1997.
- [5] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. on Computers*, 37(8):930–945, Aug. 1988.
- [6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 152–164, Pacific Grove, CA, Oct. 1991.
- [7] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *Intl. Conf. on Distributed Computing Systems*, pages 562–571, May 1998.
- [8] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. InterWeave: A Middleware System for Distributed Shared State. In *Proc. of the 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.
- [9] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Beyond S-DSM: Shared State for Distributed Systems. TR 744, Computer Science Dept., Univ. of Rochester, Mar. 2001.
- [10] G. Delamarter, S. Dwarkadas, A. Frank, and R. Stets. Portable Parallel Programming on Emerging Platforms. *Current Science Journal*, 78(7), Indian Academy of Sciences, Apr. 2000.
- [11] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Roberts, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. PerDiS: Design, Implementation, and Use of a PERSistent DIstributed Store. Research Report 3525, INRIA, Rocquencourt, France, Oct. 1998.
- [12] R. Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. In *Proc. of the 12th ACM Symp. on Principles of Distributed Computing*, Ithaca, NY, Aug. 1993.
- [13] A. S. Grimshaw and W. A. Wulf. Legion—A View from 50,000 Feet. In *Proc. of the 5th Intl. Symp. on High Performance Distributed Computing*, Aug. 1996.
- [14] R. Kordale, M. Ahamad, and M. Devarakonda. Object Caching in a CORBA Compliant System. *Computing Systems*, 9(4):377–404, Fall 1996.
- [15] S. Parthasarathy and S. Dwarkadas. InterAct: Virtual Sharing for Interactive Client-Server Applications. In *4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.
- [16] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and Interactive Sequence Mining. In *Intl. Conf. on Information and Knowledge Management*, Nov. 1999.
- [17] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architectures*, Newport, RI, June 1997.
- [18] R. Srikant and R. Agrawal. Mining Sequential Patterns. IBM Research Report RJ9910, IBM Almaden Research Center, Oct. 1994. Expanded version of paper presented at the Intl. Conf. on Data Engineering, Taipei, Taiwan, Mar. 1995.
- [19] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, St. Malo, France, Oct. 1997.
- [20] R. Stets, S. Dwarkadas, L. I. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order, Broadcast, and Remote-Write Capability on Network-Based Shared Memory Computing. In *Proc. of the 6th Intl. Symp. on High Performance Computer Architecture*, Toulouse, France, Jan. 2000.
- [21] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Support for Machine and Language Heterogeneity in a Distributed Shared State System. Submitted for publication, May 2002. Expanded version available as TR 783, Computer Science Dept., Univ. of Rochester.
- [22] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. In *IEEE Concurrency*, pages 70–78, Jan.-Mar. 1999.
- [23] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *International Workshop on Object Orientation in Operating Systems*, page 244ff, Paris, France, Sept. 1992.
- [24] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th Symp. on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.
- [25] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. In *IEEE Trans. on Parallel and Distributed Systems*, pages 540–554, 1992.