Beyond S-DSM: Shared State for Distributed Systems *

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

Technical Report #744

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

March 2001

InterWeave is a distributed middleware system that attempts to do for computer programs what the World Wide Web did for human beings: make it dramatically simpler to share information across the Internet. Specifically, InterWeave allows processes written in multiple languages, running on heterogeneous machines, to share arbitrary typed data structures as if they resided in local memory. In C, operations on shared data, including pointers, take precisely the same form as operations on non-shared data. Sharing at all levels is supported seamlessly—InterWeave can accommodate hardware coherence and consistency within multiprocessors (*level-1* sharing), software distributed shared memory (SDSM) within tightly coupled clusters (*level-2* sharing), and version-based coherence and consistency across the Internet (*level-3* sharing). 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.

We discuss the implementation of InterWeave in some detail, with a particular emphasis on memory management; coherence and consistency; and communication and heterogeneity. We then evaluate the performance and usability of the system. Anecdotal evidence suggests that the InterWeave prototype significantly simplifies the construction of important distributed applications. Quantitative evidence demonstrates that it achieves this simplification at acceptably modest cost.

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. Most of these applications rely, at least in the abstract, on some notion of distributed shared state. Access to this shared state can be accomplished either by moving the process to the data or moving the data to the process. Either option may

---

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

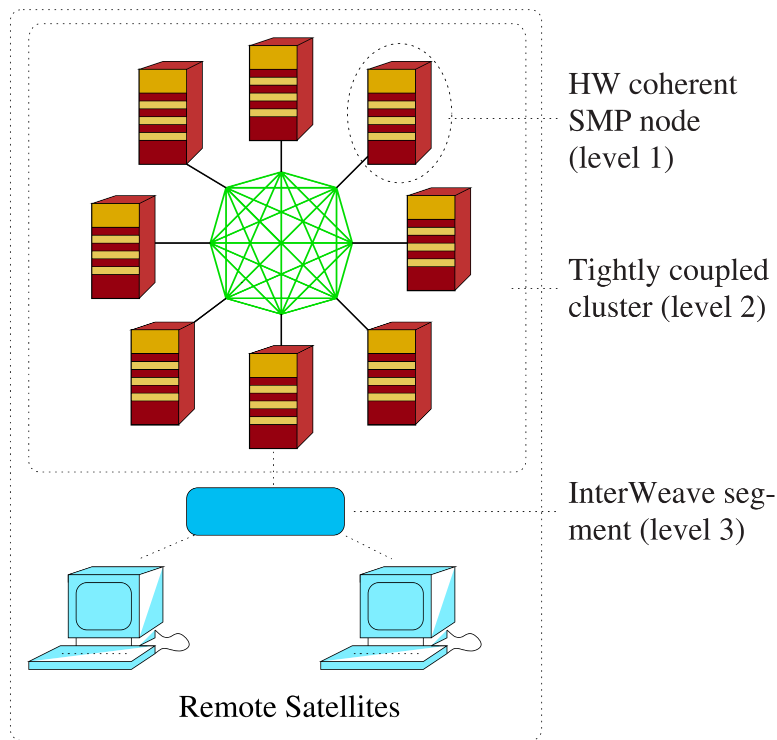

Figure 1: InterWeave's target environment.

make sense from a performance point of view, depending on the amounts of data and computation involved, the feasibility of migration, and the frequency of data updates.

The first option—move the process to the data—corresponds to remote procedure call or remote method invocation, and is supported by widely available production-quality systems. The second option—move the data to the process—is not as well understood. It still tends to be achieved through special-purpose, application-specific message-passing protocols. The creation of these protocols is a time-consuming, tedious, and error-prone activity. It is complicated by the need, for performance reasons, to cache copies of data at multiple locations, and to keep those copies consistent in the face of distributed updates.

In order to support applications with distributed shared state, we are developing a system, known as InterWeave, that allows the programmer to map shared segments into program components regardless of location or machine type. Once shared segments have been mapped, InterWeave can support hardware coherence and consistency within multiprocessors (*level-1* sharing), software distributed shared memory (SDSM) within tightly coupled clusters (*level-2* sharing), and version-based coherence and consistency across the Internet (*level-3* sharing); see Figure 1.

At the third level, each segment in InterWeave evolves through a series of consistent versions. When beginning a read-only critical section on a given 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 among segments, to avoid causality loops, we invalidate mutually-inconsistent versions of other segments,

using a novel hashing mechanism that captures the history of each segment in a bounded amount of space. SDSM-like twins and diffs allow us to update stale segments economically.

Like CORBA [26] and many older RPC systems, InterWeave employs a type system based on a machine- and language-independent interface description language, in our case Sun XDR [40]. Using knowledge of data types, InterWeave then ensures that the version of a segment cached by a given process is appropriate to the process's language and machine architecture. When transmitting data between machines, we convert to and from a standard wire format. We also swizzle pointers [45] in order to provide address independence at minimal loss in performance by representing references to data currently cached on the local machine as machine addresses. We also allow programs to organize dynamically-allocated data within a segment in different ways on different machines, for the sake of spatial locality.

We describe the design of InterWeave in more detail in Section 2, covering synchronization, coherence, consistency, heterogeneity, and integration with existing hardware and software shared memory. Our initial implementation is then described in Section 3, with performance results 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 unit of sharing in InterWeave is a self-descriptive data segment within which programs allocate strongly typed blocks of memory. 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 number/name and offset (delimited by pound signs), we obtain a machine-independent pointer (*MIP*): "`iwtp://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. To create and initialize a segment in C, we execute the following calls:

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

Every segment is managed by an InterWeave server at the IP address indicated in the segment's URL. Different segments may be managed by different servers. Assuming appropriate access rights, the `IW_create_segment` call communicates with the appropriate server to create an uninitialized segment, and allocates space to hold (the initial portion of) a local cached copy of that segment in the caller's address space. The handle returned by `IW_create_segment` is an opaque, machine-dependent type that may be passed to `IW_malloc`, along with a type descriptor generated by our XDR compiler. The copy of a segment cached by a given process need not necessarily be contiguous in the application's virtual address space, so long as individually `malloc`ed blocks are contiguous; the InterWeave library can expand a segment as needed using unrelated address ranges.

Once a segment has been initialized, a process can create a MIP that points to an arbitrary location within one of its allocated blocks:

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

This MIP can then be passed to another process through a message, a file, or even console I/O. 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. The mechanism used to specify and verify access rights is still under development.

Any given segment A may contain pointers to data in some other segment B. The pointer-swizzling and data-conversion mechanisms described in Section 2.3 below ensure that such pointers will be valid local machine addresses, and may freely be dereferenced. It remains the programmer's responsibility to ensure that segments are accessed only under the protection of reader-writer locks. To assist in this task, InterWeave allows the programmer to identify the segment in which the datum referenced by a pointer resides, and to determine whether that segment is already locked:

```
IW_handle_t h = IW_get_handle(p);
IW_lock_status s = IW_get_lock_status(h);
```

Much of the time we expect that programmers will know, because of application semantics, that pointers about to be dereferenced refer to data in segments that are already locked.

2.1 Coherence

Given the comparatively high and variable latencies of even local-area networks, traditional hardware-inspired consistency models are unlikely to admit good performance in a distributed environment. Even the most relaxed of these models, release consistency, guarantees a coherent view of *all* shared data among *all* processes at synchronization points, resulting in significant amounts of communication. Fortunately, processes in distributed applications can often accept a significantly more relaxed—and hence less communication-intensive—notion of consistency. 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.

Coherence in InterWeave is based on the notion that segments move over time through a series of internally consistent states, under the protection of reader-writer locks. When writing a segment, a process must have exclusive access to the most recent version (we do not support branching histories). When reading a segment, however, the most recent version may not be required. 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 [37] 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 segment is 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. An event mechanism, not described here, allows a process to receive asynchronous notification of changes to segments it would not otherwise have locked.

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 3.4, often allows the implementation to avoid communication with the server when updates are not required. Twin and diff operations [7], extended to accommodate heterogeneous data formats (Section 2.3), allow the implementation to perform an update in time proportional to the fraction of the data that has changed.

Unless otherwise specified, newly-created segments employ Full coherence. The creator of a segment can specify an alternative default if desired. An individual process may also establish its own default for its own lock operations, 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 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, 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 noted in Section 1, an SDSM-style "level-2" sharing system can play the role of a single node at level 3. A 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. The runtime system guarantees 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.

2.2 Consistency

Unless specifically handled, 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_i refer to version i of segment A. If B_j was created using information found in A_i, then previous versions of A are causally incompatible with B_j; a process that wants to use B_j (and that wants to respect causality) should invalidate any cached segment versions that predate the versions on which B_j depends.

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 3.3 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 in 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.

2.3 Heterogeneity

To accommodate a variety of machine architectures, remote procedure call systems usually incorporate a language- and machine-independent notation to describe the types of parameters, together with a stub compiler that automatically translates to and from a universal "wire format". Any system for distributed shared state must provide a similar level of support for heterogeneity.

Blocks allocated within segments in InterWeave must have types defined using Sun's XDR data description language [40]. This notation is rich enough to describe arbitrarily complex combinations of primitive types, arrays, records, pointers, and strings. InterWeave guarantees that each type will have the appropriate machine-specific byte order, alignment, etc. in locally cached copies of segments. Pointers, in particular, appear as local machine addresses, and can safely be dereferenced even when they point to data in another segment.

When asked to produce output for C on a given machine architecture, our XDR compiler generates a `.h` file containing C type declarations and a `.c` file containing type descriptors in the form of C initialized variables. The descriptors describe the layout of the types on the specified machine. They must be registered with the InterWeave library at program startup, and passed to each call to `IW_malloc`. These conventions ensure that the library will be able to translate to and from wire format when communicating with a server.

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 functionality of each module is divided between the server process and the client library. We discuss the relevant features of each of these modules below.

3.1 Memory Management and Segment Metadata

As described in Section 2, InterWeave presents the programmer with two granularities of shared data: *segments* and *blocks*. A segment is a logical heap in which blocks can be allocated and freed. A block is a contiguous section of memory allocated in a segment in response to an `IW_malloc()` call. 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. A segment is a named collection of blocks. 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.

Internally, in addition to segments and blocks, an InterWeave client keeps track of *subsegments*. Subsegments are contiguous regions of memory that comprise the local copy of a segment. The subsegments of a given segment need not necessarily be contiguous with one another (and in general are not). Subsegments support blocks of arbitrary size, and allow segments to grow over time, but ensure that a given virtual memory page contains data from only one segment.

An InterWeave client manages its own heap area, rather than relying on the standard C library `malloc()`. The InterWeave heap routines manage subsegments, and maintain a variety of bookkeeping information. Among other things, this information includes a collection of balanced search trees that allow InterWeave to quickly locate blocks by name, serial number, or address, to support the translation of MIPs into local pointers and vice versa.

Figure 2 illustrates the organization of memory into subsegments, blocks, and free space. The segment table has exactly one entry for each segment being cached by the client in local memory. It is organized as a hash table, keyed by segment name. In addition to the segment name, each entry in the table includes four pointers: one for the first subsegment that belongs to that segment, one for the first free space in the segment, and two for a pair of balanced trees containing the segment's blocks. One tree is sorted by block serial number, the other by block symbolic name; together they support translation from MIPs to local pointers. The segment table entry may also include a cached TCP connection over which to reach the server.

Each block in a subsegment begins with a header containing the size of the block, a pointer to a type descriptor, a serial number, and an optional symbolic block name. Free space within a segment is kept on a linked list, with a head pointer in the segment table. Allocation is currently first-fit. To allow a deallocated block to be coalesced with its neighbor(s), if free, all blocks have a footer (not shown in Figure 2) that indicates whether that block is free or not and, if it is, where it starts. In an attempt to maximize locality of reference, the code that creates the initial copy of a segment at a given client places blocks in contiguous locations if they were last modified (by some other client) during a single critical section.

To accommodate reference types, InterWeave relies on pointer swizzling [45]. Briefly, swizzling uses type descriptors to find all (machine-independent) pointers within a newly-cached or updated

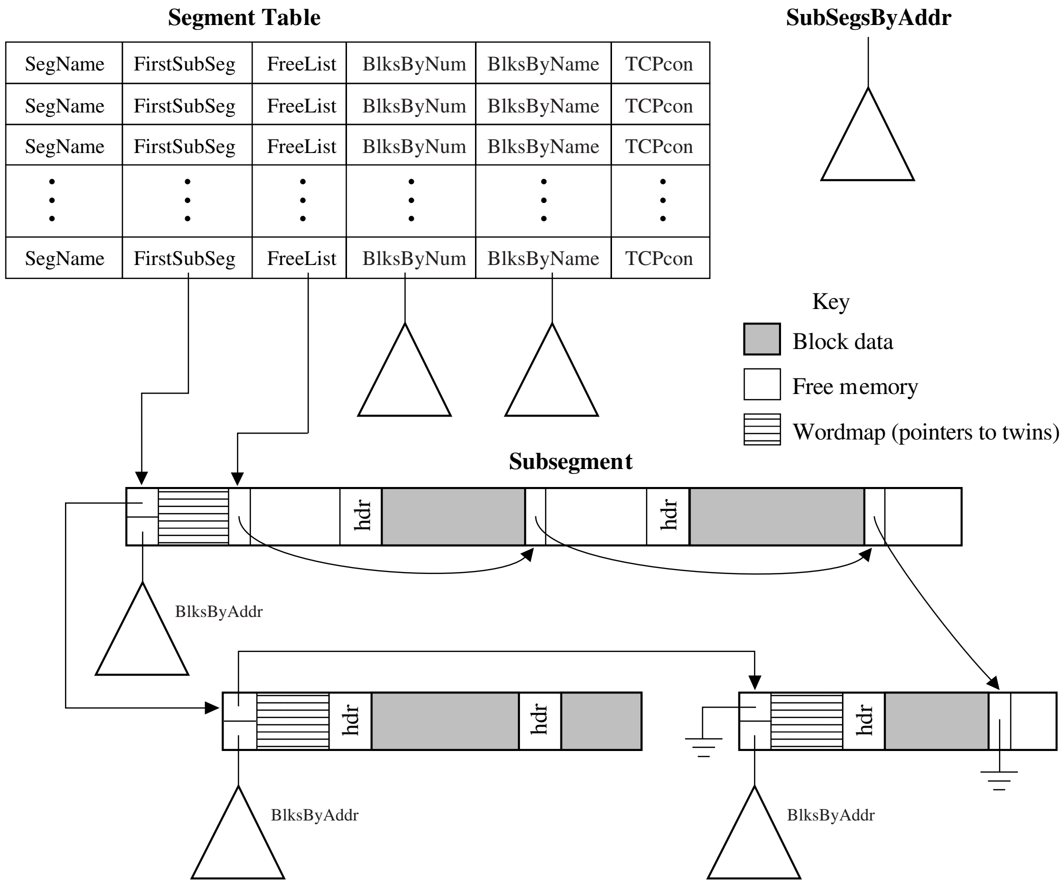

Figure 2: Simplified view of InterWeave data structures: the segment table, subsegments, and blocks within segments. Type descriptors, pointers from balanced trees to blocks and subsegments, and footers of blocks and free space are not shown.

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.

An InterWeave server keeps track of segments, blocks, and *subblocks*. The blocks of a given segment are organized into a pair of balanced trees, one sorted by serial number, the other by version number. Each subblock comprises a small contiguous group of primitive data elements from the same block, on the order of a cache line in total size. Subblocks and the version number tree are used by the coherence protocol, as described in Section 3.3. To avoid an extra level of translation, the server stores both data and type descriptors in wire format. In order to avoid unnecessary data relocation, machine-independent pointers and strings are stored separately from their blocks, since they can be of variable size.

3.2 Detecting Modifications and Translating to Wire Format

When a process acquires a write lock on a given segment, the InterWeave library asks the operating system to write protect the pages that comprise the various subsegments of 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*, 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. More specifically, if the fault occurs in page i of subsegment j, the handler places a pointer to the twin in the ith entry of a structure called the *wordmap*, located in j's header (see Figure 2). A global balanced tree of subsegments, sorted by memory address, makes it easy for the handler to determine i and j. Together, the wordmaps and the linked list of subsegments in a given segment allow InterWeave to quickly determine which pages need to be diffed when the coherence protocol needs to send an update to the server.

When a process releases a write lock, the library scans the list of subsegments of each segment and the wordmap within each subsegment. When it identifies a group of contiguous modified pages, it performs a word-by-word diff of the pages and their twins. In order to convert this diff into machine-independent wire format, the diffing routine must express all changes in terms of segments, blocks, and primitive data unit offsets, rather than pages and bytes. It must also have access to type descriptors, in order to compensate for local byte order and alignment, and in order to swizzle pointers.

To make descriptors easy to find, each subsegment contains a root pointer for a balanced tree of the subsegment's blocks, sorted by memory address. The diffing routine uses the balanced tree to identify the block in which the first modified word lies. It then scans blocks linearly, converting each to wire format. The conversion is driven by the type descriptor pointers found at the beginnings of blocks. When it runs off the end of the last contiguous modified page, the diffing routine returns to the wordmap and subsegment list to find the next group of contiguous modified pages. By looking only at modified pages, and only at allocated blocks within those pages, we avoid unnecessary diffing.

When a client acquires a lock and determines that its copy of the segment is not recent enough, the server builds a diff that describes the data that have changed between the client's outdated copy and the master copy at the server. Further details appear in Section 3.3. To convert the serial

numbers employed in wire format to local machine addresses, the client traverses the balanced tree of blocks, sorted by serial number, that is maintained for every segment.

Both translations between local and wire format—for server updates at write lock release and for client updates at lock acquisition—are driven by type descriptors. InterWeave requires that these descriptors be registered with the client library at process startup time. (Simply remembering the descriptors that are passed to `IW_malloc` is not enough: a process may use shared data of a given type even if it never actually *allocates* data of that type itself.) For the sake of programming convenience, a special linker cooperates with the InterWeave XDR compiler to insert the registration calls into the object code automatically; the programmer need not write them.

The content of each such descriptor specifies the substructure and machine-specific layout of its type. For primitive types (integers, doubles, etc.) there is a single pre-defined code. For other types there is a code indicating either an array, a record, or a pointer, together with pointer(s) that recursively identify the descriptor(s) for the array element type, record field type(s), or pointed-at type.

Like blocks, type descriptors have segment-specific serial numbers, which the server and client use in wire-format messages. A given type descriptor may have different serial numbers in different segments. Per-segment arrays and hash tables maintained by the client library map back and forth between serial numbers and pointers to local, machine-specific descriptors. An additional, global, hash table maps wire format descriptors to addresses of local descriptors, if any. When an update message from the server announces the creation of a block with a type that has not appeared in the segment before, the global hash table allows InterWeave to tell whether the type is one for which the client has a local descriptor. (If there is no such descriptor, then the new block can never be referenced by the client's code, and can therefore be ignored.)

3.3 Coherence and Consistency

Each server maintains an up-to-date copy of each segment it serves, and controls access to the segments. 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. The server also remembers, for each block, the version number in which the block was created. Finally, the server maintains a list of the serial numbers of deleted blocks along with the versions in which they were deleted. The creation version number allows the server to tell, when sending a segment update to a client, whether the metadata for a given block needs to be sent in addition to the data. When *malloc*ing a new block, a client can use any available serial number. Any client with an old copy of the segment in which the number was used for a different block will receive new metadata from the server the next time it obtains an update.

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 below.) If an update is required, the client sends the server the (out-of-date) version number of the local copy. The server then traverses the tree of blocks sorted by version number (and, within each modified block, a submap of version numbers for subblocks) to identify the data that has changed since the

last update to this client. Having identified this data, it constructs a wire-format diff and returns it to the client.

3.3.1 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.

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. With care, this scheme can be designed to safely accommodate roll-over of the values within timestamps, and to reduce the chance that hash collisions will cause repeated extraneous invalidations of a segment that seldom changes.

3.3.2 Integration with 2-Level System

As mentioned in Section 1, InterWeave allows seamless integration with more tightly-coupled sharing at the hardware and SDSM level. In our implementation, we use Cashmere-2L [39] as a representative SDSM system that takes advantage of hardware shared memory within multiprocessor nodes in addition to providing the same abstraction in software across nodes. 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 then kept in cluster-wide shared memory.

In the current implementation, we designate a single node within the cluster to be the segment's *manager* node. All interactions with 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 creates a level-3 twin for a page if it experiences a write fault, if it is the level-2 home node for the page and it receives a level-2 diff from another node in the cluster, or if it receives a write notice from another node in the cluster and must invalidate the page (see [39] for details on the Cashmere-2L implementation). 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.

3.3.3 Support for Diff Coherence

Among the coherence models built into InterWeave, only Diff coherence requires support at the server beyond the version numbers associated with blocks and subblocks. For each client using Diff coherence, the server must track the percentage of the segment that has been modified since the last update sent to the client. To minimize the cost of this tracking, the server conservatively assumes that all updates are to independent portions of the segment. It adds the sizes of these updates into a single counter. When the counter exceeds the specified fraction of the total size of the segment (which the server also tracks), the server concludes that the client's copy is no longer recent enough.

3.4 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 or events. 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). Event notifications from a server to a client are caught with a SIGIO handler that in turn invokes any client-registered handler.

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. 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 recently-requested diffs, to avoid redundant overhead when several clients that have cached the same version of the segment need to be updated. Finally, as in the TreadMarks SDSM system [3], a client that repeatedly modifies most of the data in a

	Alpha	Sun	PC
twin creation	64.6 (8K)	73.3 (8K)	32.1 (4K)
page fault	18.0	122	10.1
mprotect	3.79	11.6	2.42
IW write fault	88.6	209	46.7
IW_malloc	2.22–4.64	3.42–5.63	3.03–4.99
IW_free	1.37–2.70	2.64–4.61	2.16–4.63
IW_mip_to_ptr	1.71–4.18	3.49–5.05	4.96–6.81
IW_ptr_to_mip	2.02–4.19	3.15–5.22	5.93–7.98
roundtrip TCP/IP	150 (200)	233	333

Table 1: Basic operation costs (all times in μsecs).

segment will switch to a mode in which it simply transmits the whole segment to the server at every write lock release. This mode eliminates the overhead of `mprotect`s, page faults, and the creation of twins and diffs. The switch occurs when the client finds that the size of a newly created diff is at least 75% of the size of the segment itself; this value strikes a balance between communication cost and the other overheads of twinning and diffing. Periodically afterwards (at linearly increasing random intervals), the client will switch back to diffing mode to verify the size of current modifications.

4 Performance Results

4.1 Platforms and Microbenchmarks

In order to demonstrate and evaluate the various features of InterWeave, we used a collection of machines of different types. The high-end cluster we use for our parallel applications is an AlphaServer system. Each node is an AlphaServer 4100 5/600, 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 [13] system area network, which is used for tightly-coupled sharing. Connection to the local area network is via TCP/IP over Fast Ethernet. We also use Sun Ultra 5 workstations with 400 MHz Sparc v9 processors with 128 MB of memory, running SunOS 5.7, and 333 MHz Celeron PCs with 256 MB of memory, running Linux 6.2.

Table 1 provides statistics on the costs of various basic operations on each machine type. Twin creation is the cost of copying an uncached page on each platform (the operating system (OS) page size is indicated in parentheses). The page fault time is the OS cost to transfer execution from the faulting instruction to the `SIGSEGV` handler and to return when the handler is complete; this cost is particularly high on SunOS. The mprotect cost is the average cost per page for changing access permissions on the page. The IW write fault time is the total overhead incurred by the InterWeave system on the first write access after a write lock acquire to any page belonging to an InterWeave segment. This time includes the page fault, twin creation, and mprotect times, as well as some time to search and update InterWeave metadata.

	Alpha	Sun	PC
per 4-byte word, mixed			
collect block	.372	.582	.752
collect diff	.490	.798	1.05
apply block	.200	.320	.397
apply diff	.303	.449	.514
per 4-byte integer			
collect block	.024	.040	.064
collect diff	.280	.568	.428
apply block	.044	.044	.072
apply diff	.040	.044	.072
per pointer			
collect block	4.40	7.83	9.07
collect diff	4.54	8.44	9.39
apply block	2.04	3.05	4.12
apply diff	2.08	3.27	4.17

Table 2: Data translation costs.

`IW_malloc`, `IW_free`, `IW_mip_to_ptr`, and `IW_ptr_to_mip` indicate the time taken per operation. The smaller number represents the average time for a 256-block segment; the larger number represents the average time for a segment with 1M blocks. In both cases the block contains a single integer. The figures differ because of traversal costs in balanced trees.

Also shown are average roundtrip TCP/IP times, as a measure of underlying platform overhead. These times are for communication from an Alpha process to another process within the same Alpha node, on a different Alpha node (via Ethernet; indicated in parentheses), and between an Alpha process and a Sun or PC, respectively.

Table 2 shows the range of costs associated with translation to (*collect*) and from (*apply*) wire format. The first group of numbers is the cost per word (4 bytes) for a "typical" data structure containing integer, double, float, string, and pointer types. The "block" lines cover the case of newly created blocks, which are transmitted in full; the "diff" lines cover the case in which only the modified words are transmitted. The second and third groups of numbers are for the extreme cases of a simple integer (4 bytes) array, for which byte order is the only potential complexity, and pointers (8 bytes on the Alphas, 4bytes on the Suns and PCs), which require swizzling.

The current implementation of data translation involves recursive analysis of type descriptors. By comparing the performance of InterWeave's translation library to hand-generated translation code, we determined that an additional 17% of the translation overhead could be eliminated if we were to modify our XDR compiler to embed knowledge of type formats directly in the translation code.

4.2 Critical Section Scalability

Figure 3 shows the time per write lock acquire/release as the number of clients of a segment increases. In the lower curve only one word of data is actually modified; in the upper curve each

client modifies every element of a 1024-word array. All clients are on different Alpha processors, the server is on a Sun, and communication is via Fast Ethernet. Because writer critical sections are serialized, the server is able to accommodate additional clients at little additional cost, and the time per critical section (calculated as the total execution time divided by the total number of acquires performed) stays relatively constant. The jump in time from 1 to 2 clients reflects the loss of *exclusive* mode, which avoids communication with the server in the absence of sharing.

Figure 4 shows the time per read lock acquire assuming Full coherence. In this experiment, a single client acquires a write lock and modifies either 1 or 1024 integers. All clients then acquire a read lock on the data. The time per reader critical section (calculated as the total execution time, minus the time spent in extraneous code, divided by the number of read lock acquires per client) increases with the number of clients, reflecting serialization of service to clients by the (uniprocessor) server. The use of multicast, where available, might produce a flatter curve.

Figure 5 demonstrates the effectiveness of Diff coherence. In this experiment there are 16 clients. One client modifies 1% of the data in each of a long series of writer critical sections. After each update all 16 clients acquire a read lock under Diff coherence, specifying different thresholds for tolerable percentage of data changed. The y-axis represents the corresponding time to acquire and release each read lock, calculated in the same manner as for Figure 4. The lower three curves show significant reductions in communication overhead and, consequently, average acquire/release overhead.

The differences among these last three curves illustrate the benefit of the adaptive protocol described in Section 3.4. When the ratio of read acquires that do not require data communication to write acquires is ≥ 1, the use of the notification mode avoids the poll messages that would otherwise be sent to determine if the segment needs to be updated at the client. When the ratio of read acquires that do not require data communication to write acquires is < 1, it is desirable to switch back to polling mode, especially with a large number of clients, in order to avoid extra messages and unnecessary overhead at the server. With diff coherence, as the diff parameter is increased, the number of read lock acquires that result in no data being transmitted between server and client increases. Dynamically switching among polling and notification allows a general-purpose protocol to closely approximate the performance of pure notification in this experiment. In this experiment there is no measurable difference among polling, notification, and adaptive variants of Full coherence; they are all represented by the “Full-adaptive” curve.

4.3 API Ease-of-Use

Our calendar program illustrates the ease with which distributed applications can be prototyped with InterWeave. 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.

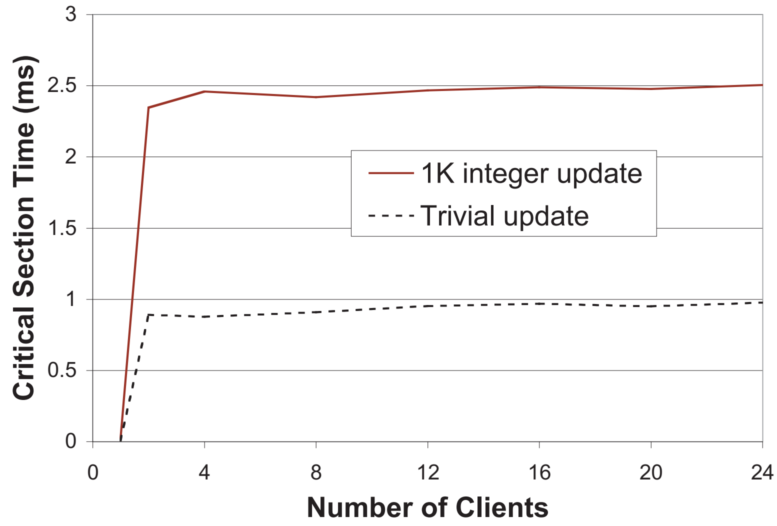

Figure 3: Sequential cost of acquiring a write lock.

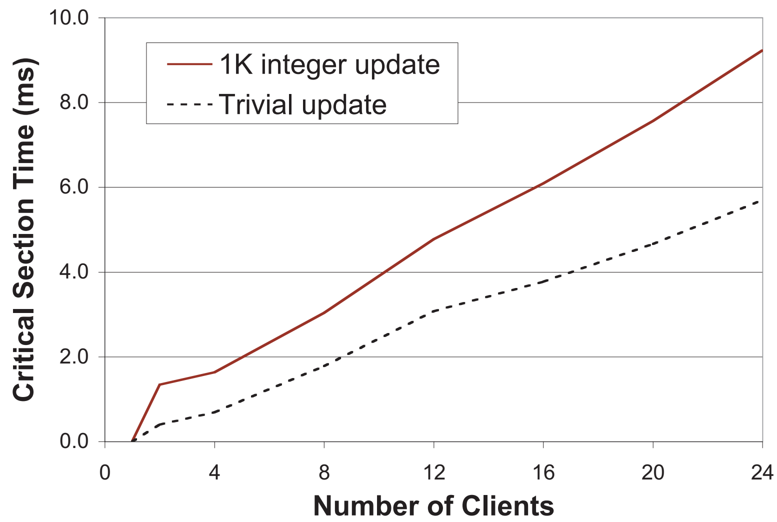

Figure 4: Cost for a given number of clients to acquire a read lock concurrently.

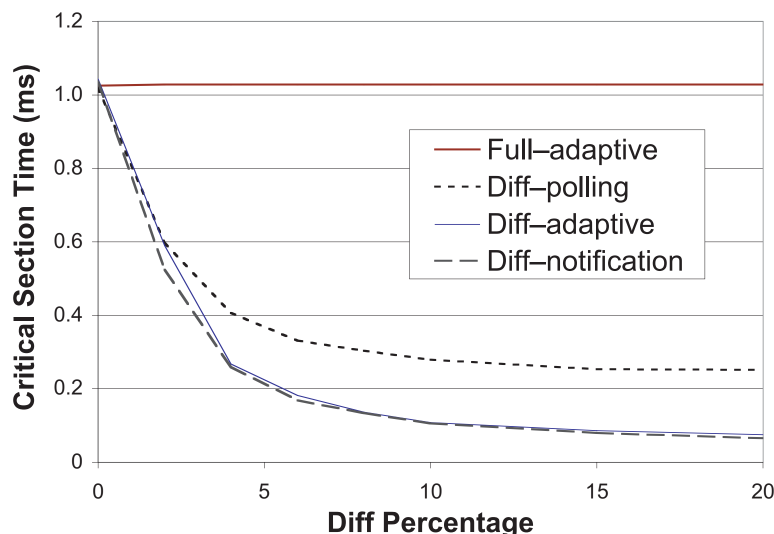

Figure 5: Comparison of full and diff coherence.

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 sockets-based code, the InterWeave calendar program has no message buffers, no marshalling and unmarshalling 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.

4.4 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 of a remote visualization of the Splash-2 [46] Barnes-Hut simulation. The simulation runs on a 4-node, 16-processor AlphaServer system, and repeatedly computes new positions for 16,384 bodies. These positions may be shared with a remote visualization satellite via an InterWeave segment. The simulator uses a write lock to update the shared segment, while the satellite uses a relaxed read lock with temporal coherence to obtain an effective frame rate of 15 frames per second.

Under human direction, the visualization satellite can also steer the application by acquiring a write lock and changing a body's data.

When we combine the high performance second level shared memory (Cashmere) with the third level shared memory (InterWeave), it would be ideal if there were no degradation in the performance of the second level system. To see how closely we approach this ideal, we linked the application with the InterWeave library, but ran it without connecting to a visualization satellite. Communication with the server running on another Alpha node was via TCP/IP over Fast Ethernet. Relatively little communication occurs in the absence of a satellite, due to the *exclusive mode* optimization described in Section 3.4.

Execution times for the no-satellite experiment appear in Figure 6. Each bar gives aggregate wall-clock time for ten iteration steps. Each pair of bars is for a different number of processors (the configuration specifies the number of nodes and the total number of processors used, implying that *processors/nodes* processors per node were used), with the one on the left for the standard Cashmere system with no external communication and the one on the right for Cashmere linked with the InterWeave library and communicating with a server. The right-hand bars are subdivided to identify the overhead due to running the third-level protocol code.

We also measured the simulator's performance when communicating with a single satellite. Specifically, we compared execution times using InterWeave to those obtained by augmenting user-level code with explicit TCP/IP messages to communicate with the satellite (directly, without a server), and then running the result on the standard Cashmere system. Figure 7 presents the resulting execution time. In all cases the satellite was running on another Alpha node, communicating with the cluster and server, if any, via TCP/IP over Fast Ethernet. We have again subdivided execution time, this time to separate out both communication and (for the right-hand bars) InterWeave protocol overhead and wire format translation overhead. The overhead of the InterWeave protocol itself remains relatively small. For this particular sharing scenario, much of the shared data is modified at every interval. Hence, the dynamic use of diffing is able to eliminate unnecessary diffing and twinning overhead. Meta-data communication in InterWeave adds less than 1% to the total data communicated.

A key advantage of the InterWeave version of the visualization program is that the simulation need not be aware of the number of satellites or the frequency of sharing. In the version of the application that uses hand-written message passing, this knowledge is embedded in application source code.

4.5 Coherence Model Evaluation

We use a datamining application (described in [29]) 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 increase. As updates arrive the server incrementally maintains

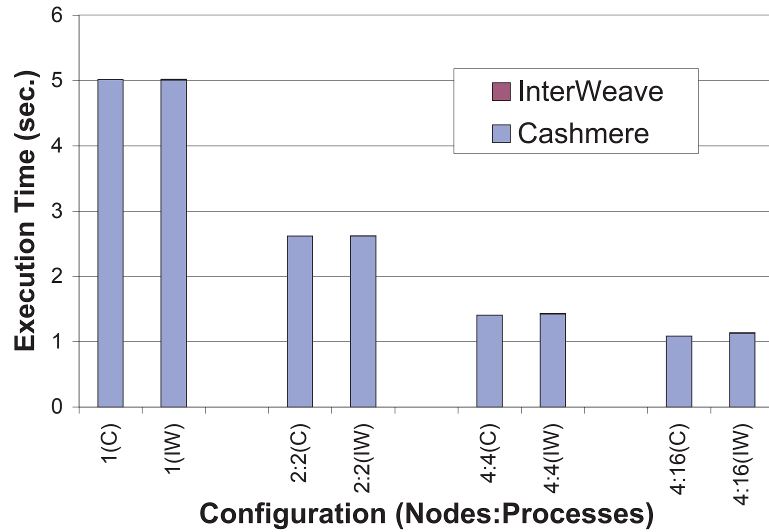

Figure 6: Overhead of InterWeave library for the Barnes-Hut application, without a visualization satellite.

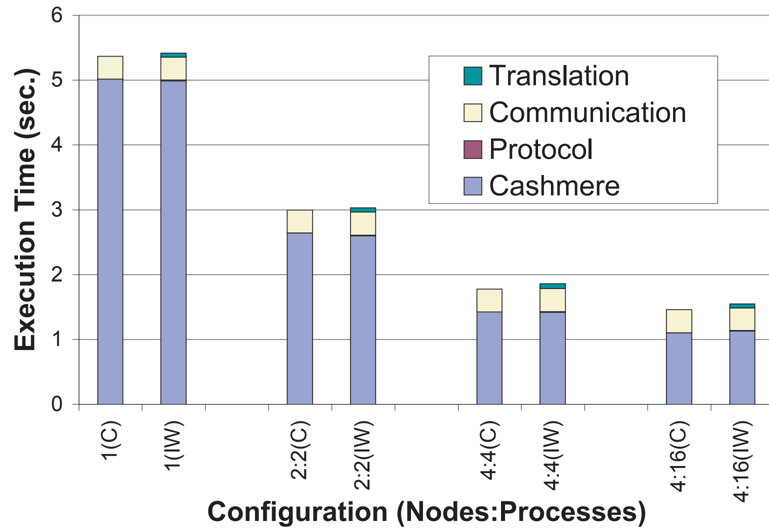

Figure 7: Overhead of InterWeave library and communication during Barnes-Hut remote visualization.

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 [38]. The summary structure is initially generated using half this 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 user interface 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 differ coherence will suffice, and can dramatically reduce communication overhead. To illustrate 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.

In Figure 8 the upper curve represents the network bandwidth required to send a complete copy of the summary data structure to one client after every incremental update (the experiment was set up to ensure a read lock acquire after every modification). The lower curve represents the bandwidth to send only diffs. The savings is as high as 71%.

In Figure 9 the upper curve is a copy of the lower curve from Figure 8. Additional curves represent the network bandwidth required with relaxed coherence. Using Diff coherence with a threshold of 30%, we see a savings of about a third. Using Delta coherence with a threshold of 4 updates, we see a savings of almost one half.

In Figure 10 we plot the average latency to acquire a client read lock as the number of clients and the coherence model varies. In this graph, the rate of modification of the summary structure is 0.11/sec. Each client acquires the summary structure at an approximate rate of 2/sec (as measured when there is a single client). With full coherence the time per lock increases with the number of clients, just as it did in Figure 4. With relaxed coherence it still increases, but with a much lower constant factor. Since a significant number of the nodes are modified (with potentially a small change in their values) with each new version, delta coherence is able to reduce more communication than diff coherence. The choice of coherence model would depend on whether the magnitude of change in value of each node versus the number of changed nodes was more or less significant to the mining query. These numbers are representative of the improvement in performance possible using application-specific knowledge about the tolerance for relaxed coherence. As demonstrated in [27], for many data mining scenarios, the degradation in result quality with the use of relaxed coherence can be negligible.

5 Related Work

InterWeave finds context in an enormous body of related work—far too much to document thoroughly here. We focus here on some of the most relevant systems in the literature that support replication.

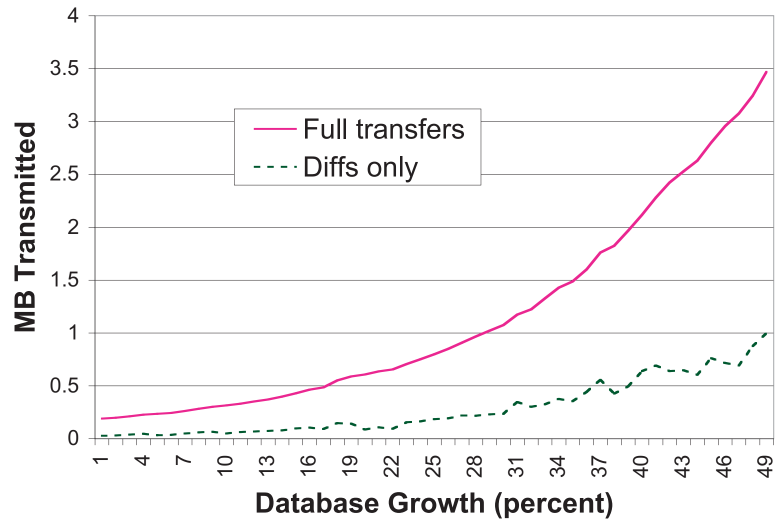

Figure 8: Sequence mining: bandwidth required per read lock acquire.

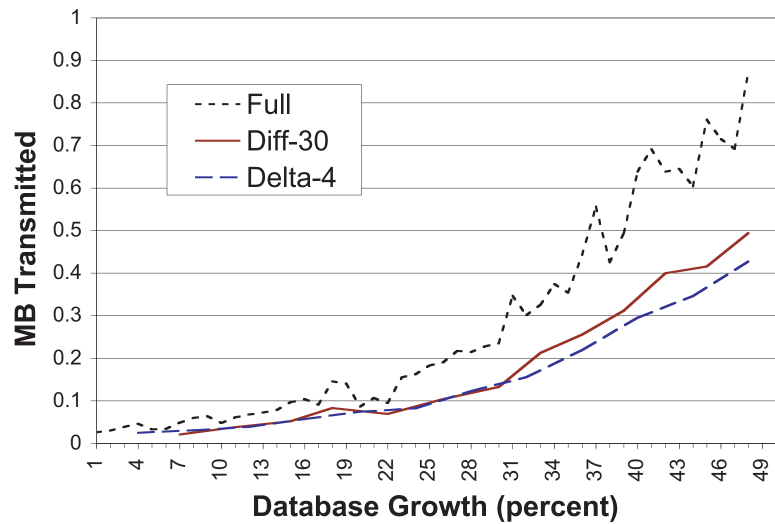

Figure 9: Sequence mining: bandwidth required under different coherence models.

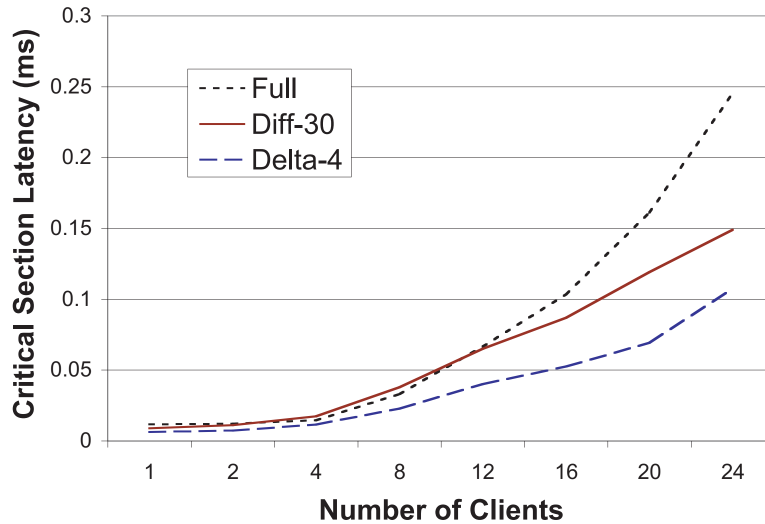

Figure 10: Sequence mining: latency per read lock acquire.

Most systems for distributed shared state enforce a strongly object-oriented programming model. Some, such as Emerald [21], Argus [24], Ada [20], and ORCA [42], take the form of an explicitly distributed programming language. Some, notably Amber [10] and its successor, VDOM [11], are C++-specific. Many in recent years have been built on top of Java; examples include Aleph [19], Charlotte [4], Java/DSM [49], Javelin [6], JavaParty [30], JavaSpaces [41], and ShareHolder [17]. Language-independent distributed object systems include PerDiS [12], Legion [16], Globe [44], DCOM [34], and various CORBA-compliant systems [26]. Globe replicates objects for availability and fault tolerance. PerDiS and a few CORBA systems (e.g., Fresco [23]) cache objects for locality of reference. Thor [25] enforces type-safe object-oriented access to records in a heterogeneous distributed database.

At least two early software distributed shared memory (S-DSM) systems provided support for heterogeneous machine types. Toronto's Mermaid system [50] 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.

Khazana [8] supports distributed sharing without enforcing an object-oriented programming style. Khazana proposes a global, 128-bit address space for all the world's shared data. It does not impose any structure on that data, or attempt to translate it into locally-appropriate form. InterAct [28] is an object-based system that uses relaxed coherence models to provides distributed sharing support. InterWeave has similar notions of relaxed coherence that have been extended to include inter-segment consistency, adaptation to application communication behavior, and support for more tightly-coupled sharing.

Stampede [32] is a system designed specifically with multimedia applications in mind. A data sharing abstraction called *space-time memory* is defined, which allows processes to access a time-sequenced collection of data items easily and efficiently. One of the novel aspects of this system is the buffer management and garbage collection of this space-time memory. InterWeave focuses on providing semantics similar to hardware shared memory, and therefore supports provision of only the latest version of shared data. Deno [9] uses a decentralized currency mechanism to establish consistency and support distributed replication of objects for reliability in the presence of mobility and weak connectivity. Various Linda systems [31, 36] also provide a non-object-oriented distributed shared store.

Interface description languages date from Xerox Courier [47] and related systems of the early 1980s. Precedents for the automatic management of pointers include Herlihy's thesis work [18], LOOM [22], and the more recent "pickling" (serialization) of Java [33]. Friedman [15] and Agrawal et al. [1] have shown how to combine certain pairs of consistency models in a non-version-based system. Alonso et al. [2] present a general system for relaxed, user-controlled coherence. Yu et al. [48] describe a system in which consistency can vary continuously in three orthogonal dimensions. Khazana also proposes the use of multiple consistency models. We explore dynamically adjustable coherence in an environment that allows tightly-coupled sharing in addition to the relaxed coherence appropriate for distributed data. Many SDSM systems, including Munin [7], TreadMarks [3], and Cashmere [39], have used (machine-specific) diffs to propagate updates. Several projects, including ShareHolder, Globus [14], and WebOS [43], use URL-like names for distributed objects or files.

6 Conclusions and Future Work

Our InterWeave prototype is now up and running on a variety of hardware platforms. 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. Quantitative measurements, as described in Section 4, indicate that the overhead imposed by InterWeave is modest in comparison to the existing latency of TCP/IP. Moreover, InterWeave facilitates the use of relaxed coherence and consistency models that greatly reduce communication costs, and that are much more difficult to implement in hand-written message-passing code. InterWeave also facilitates the seamless sharing of data with more tightly-coupled clusters and SMPs.

Of course, the simplicity of InterWeave's semantics is also its principal limitation. For complex distributed applications involving untrusted code from multiple clients and vendors, we have no ambition to replace such object-based standards as CORBA and DCOM. Increasingly, however, we see a need for standards that support simpler applications with a lower learning curve and less notational "boilerplate". By analogy, InterWeave strives for the simplicity and convenience of a distributed file system, rather than the power of a distributed object-oriented database.

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, incremental interactive data mining, and human-computer collaboration in richly instrumented physical environments. With the basic features of InterWeave in place, we are also turning to several functional enhancements, including security, fault tolerance, transactional semantics, and support for streaming media. We expect to leverage the protection and security work of

others, most likely using a group-based system reminiscent of AFS [35]. Fault tolerance will be simplified by leveraging the version-based programming model: a segment can simply revert to the previous version if a client dies in the middle of an update, and versions at servers can be pushed to stable storage. Ultimately, a true transactional programming model (as opposed to simple reader-writer locks) would allow us to recover from failed operations that update multiple segments, and to implement two-phase locking to recover from deadlock or causality violations when using nested locks.

Acknowledgments

Srinivasan Parthasarathy developed the InterAct system as part of his Ph.D. research, 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] 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.

[2] 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.

[3] 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.

[4] A. Baratloo, M. Karaul, V. Keden, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Intl. Journal on Future Generation Computer Systems*, 15(5-6):559–570, 1999.

[5] R. Bisiani and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines. *IEEE Trans. on Computers*, 37(8):930–945, Aug. 1988.

[6] P. Cappello, B. O. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In *1997 ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.

[7] 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.

[8] 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.

[9] U. Cetintemel and P. J. Keleher. Light-Weight Currency Management Mechanisms in Deno. In *Proc. of the 10th IEEE Workshop on Research Issues in Data Engineering (RIDE2000)*, San Diego, CA, Feb. 2000. Award paper.

[10] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proc. of the 12th ACM Symp. on Operating Systems Principles*, pages 147–158, Litchfield Park, AZ, Dec. 1989.

[11] M. J. Feeley and H. M. Levy. Distributed Shared Memory with Versioned Objects. In *OOPSLA '92 Conf. Proc.*, pages 247–262, Vancouver, BC, Canada, Oct. 1992.

[12] 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, Implementaiton, and Use of a PERsistent DIstributed Store. Research Report 3525, INRIA, Rocquencourt, France, Oct. 1998.

[13] M. Fillo and R. B. Gillett. Architecture and Implementation of Memory Channel 2. *Digital Technical Journal*, 9(1):27–41, 1997.

[14] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[15] 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.

[16] 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.

[17] J. Harris and V. Sarkar. Lightweight Object-Oriented Shared Variables for Distributed Applications on the Internet. In *OOPSLA '98 Conf. Proc.*, pages 296–309, Vancouver, Canada, Oct. 1998.

[18] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Trans. on Programming Languages and Systems*, 4(4):527–551, Oct. 1982.

[19] M. Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *Workshop on Communication, Architecture, and Applications for*, Orlando, FL, Jan. 1999.

[20] International Organization for Standardization. Information Technology — Programming Languages — Ada. Geneva, Switzerland, 1995. ISO/IEC 8652:1995 (E). Available in hypertext at `http://www.adahome.com/rm95/`.

[21] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(1):109–133, Feb. 1988. Originally presented at the *11th ACM Symp. on Operating Systems Principles*, Nov. 1987.

[22] T. Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *OOPSLA '86 Conf. Proc.*, pages 87–106, Portland, OR, Sept. – Oct. 1986.

[23] R. Kordale, M. Ahamad, and M. Devarakonda. Object Caching in a CORBA Compliant System. *Computing Systems*, 9(4):377–404, Fall 1996.

[24] B. Liskov. Distributed Programming in Argus. *Comm. of the ACM*, 31(3):300–312, Mar. 1988.

[25] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, Montreal, PQ, Canada, June 1996.

[26] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Framingham, MA, July 1996.

[27] S. Parthasarathy and S. Dwarkadas. Shared State for Distributed Interactive Data Mining Applications. In *1st SIAM Intl. Conf. on Data Mining (SDM 2001)*, Chicago, IL, Apr. 2001.

[28] 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.

[29] 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.

[30] M. Philippsen and M. Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency — Practice and Experience*, 9(11):1125–1242, Nov. 1997.

[31] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proc. of the 21st Intl. Conf. on Software Engineering*, pages 368–377, Los Angeles, CA, May 1999.

[32] U. Ramachandran, N. Harel, R. S. Nikhil, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proc. of the 7th ACM Symp. on Principles and Practice of Parallel Programming*, pages 183–192, Atlanta, GA, May 1999.

[33] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling State in the Java System. *Computing Systems*, 9(4):291–312, Fall 1996.

[34] D. Rogerson. *Inside COM*. Microsoft Press, Redmond, Washington, Jan. 1997.

[35] M. Satyanarayanan. Integrating Security in a Large Distributed System. *ACM Trans. on Computer Systems*, 7(3):247–280, Aug. 1989.

[36] Scientific Computing Associates Inc. Virtual Shared Memory and the Paradise System for Distributed Computing. Technical Report, New Haven, CT, April 1999.

[37] 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.

[38] 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.

[39] 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.

[40] *Network Programming Guide — External Data Representation Standard: Protocol Specification*. Sun Microsystems, Inc., 1990.

[41] Sun Microsystems. JavaSpaces Specification. Palo Alto, CA, Jan. 1999.

[42] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *Computer*, 25(8):10–19, Aug. 1992.

[43] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proc. of the 7th Intl. Symp. on High Performance Distributed Computing*, Chicago, IL, July 1998.

[44] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. In *IEEE Concurrency*, pages 70–78, Jan.-Mar. 1999.

[45] 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.

[46] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[47] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Technical Report XSIS 038112, Dec. 1981.

[48] 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.

[49] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency — Practice and Experience*, 9(11), Nov. 1997.

[50] 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.