Exploiting High-level Coherence Information to Optimize Distributed Shared State *

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

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

ABSTRACT

InterWeave is a distributed middleware system that supports the sharing of strongly typed, pointer-rich data structures across a wide variety of hardware architectures, operating systems, and programming languages. As a complement to RPC/RMI, InterWeave facilitates the rapid development of maintainable code by allowing processes to access shared data using ordinary reads and writes.

Internally, InterWeave employs a variety of aggressive optimizations to obtain significant performance improvements with minimal programmer effort. In this paper, we focus on *application-specific* optimizations that exploit dynamic high-level information about an application's spatial data access patterns and the stringency of its coherence requirements. Using applications drawn from computer vision, datamining, and web proxy caching, we illustrate the specification of coherence requirements based on the (temporal) concept of "recent enough" to use, and introduce two (spatial) notions of *view*s, which allow a program to limit coherence management to the portion of a data structure actively in use. Experiments with these applications show that InterWeave can reduce their communication traffic by up to one order of magnitude with minimum effort on the part of the application programmer.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms

Design, Experimentation

*This work was supported in part by NSF grants CCR-9988361, CCR-0204344, CCR-0219848, ECS-0225413, and EIA-0080124; by DARPA/ITO under AFRL contract F29601-00-K-0182; by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460; and by equipment or financial grants from Compaq, IBM, Intel, and Sun.

PPoPP'03, June 11–13, 2003, San Diego, California, USA.

1. INTRODUCTION

As the Internet becomes increasingly central to modern computing, more and more applications are taking advantage of resources at distributed sites. Example problem domains include e-commerce, computer-supported collaborative work, multi-player games, intelligent environments, interactive data mining, and remote visualization and steering of real or simulated systems. Conceptually, most applications in these domains involve some sort of *distributed shared state*: information that has relatively static structure but mutable content, and that is needed at more than one site. To allow applications to work efficiently across high-latency, low-bandwidth links, programmers typically attempt to increase the locality of shared state through ad-hoc, application-specific caching or replication protocols built on top of RPC-based systems such as Sun RPC, Java RMI, CORBA, and .NET.

Software distributed shared memory (S-DSM) systems such as TreadMarks [3] and Cashmere [39] automate the management of shared state for local-area clusters, but they do not extend in any obvious way to geographic distribution. As a general rule, they assume that sharing processes are part of a single parallel program, running on homogeneous hardware, with communication latencies and bandwidths typical of modern local- or system-area networks, and with data lifetimes limited to that of the running program.

Our recent InterWeave system [12, 41], by contrast, caters explicitly to the needs of distributed applications. It provides programs with a global, persistent store that can be accessed with ordinary reads and writes, regardless of the machine architectures or programming languages used. As a complement to RPC, InterWeave serves to (1) eliminate hand-written code that maintains the coherence and consistency of cached data; (2) support genuine reference parameters in RPC calls, eliminating the need for overly conservative deep-copy parameters or, in many cases, for callback invocations; and (3) reduce the number of "trivial" invocations used simply to put or get data.

Unfortunately, the convenience of a global store introduces significant performance challenges. S-DSM systems work well when the temporal and spatial sharing granularity managed by the runtime matches the access pattern of the application. Without such careful matching, significant overhead may be incurred to maintain data that are not actually shared at present or, worse yet, that are *falsely* shared, e.g. as an artifact of co-location in the same page of virtual memory. This overhead is a serious problem for system-area S-DSM systems. It would be fatal for wide-area sharing.

Building on prior work in S-DSM, several recent projects have proposed mechanisms to reduce the cost of coherence. RTL [7] and InterWeave allow applications to share memory at the granularity of application-defined regions. Object View [28] allows programmers to give hints to a compiler to specify how threads use objects. The compiler then constructs caching protocols customized to application requirements. TACT [44] and InterWeave allow programmers to exploit the typically more relaxed coherence requirements of Internet applications using tunable coherence models.

A common theme in these systems is the desire for programmers to deal with coherence in high-level terms, allowing them to obtain application-specific performance optimizations without getting bogged down in the details of exactly which data are needed at particular nodes at particular points in time. We elaborate on this theme in this paper, considering high-level specifications of both the temporal (when must updates occur?) and spatial (which data must be updated?) aspects of coherence. As described in previous papers [12], we capture the temporal aspect of coherence by allowing a program to specify a predicate that determines when cached data is "recent enough" to use. For the spatial aspect of coherence (not covered in previous papers), we allow a program to specify a "view" of a shared segment that eliminates coherence management for inactive data. We consider views of both monolithic (array-based) and pointer-based data structures.

We make two key contributions to the high-level specification of coherence requirements. First, we show how highly relaxed coherence can be provided for a programming model based on ordinary reads and writes, despite the complications of heterogeneous languages and machine architectures. Second, we allow programs to adjust their spatial and temporal coherence requirements dynamically, to keep pace with changing needs. Neither language heterogeneity nor dynamic adaptation appears to be possible in compiler-based systems such as Object View.

To illustrate the need for dynamic adaptation of coherence requirements, consider an intelligent environment application, currently under development in our department [34, 35]. Cameras mounted throughout an office space monitor a common area from several vantage points. The cameras work cooperatively to discover objects by detecting events that are simultaneously observed by multiple cameras. Figure 1 gives an example of discovering objects with three cameras. Each camera is served by a computing node, and the nodes are connected by a local-area network. Each node stores captured images locally as an *image cube* (X × Y × t) and scans them for events of interest. For instance, a color change in an image region may indicate a moving object.

When interpreting an event, a given node enhances its understanding of what occurred by scanning images captured by other nodes. For non-critical tasks such as object tracking (*e.g., where is my coffee mug?*), images from other nodes may be "recent enough" to use even when they are seconds out of date. For more demanding tasks (e.g. robotic manipulation), temporal coherence requirements may become significantly tighter. Similarly, because events of interest occur in subregions of images, it suffices to share only the "interesting" image areas at any given time. While an entire image may logically be shared, the *actual* sharing varies dynamically according to the activity in the environment being observed.

The unit of sharing in InterWeave is a self-descriptive *segment* (a heap) within which programs allocate strongly typed *blocks* of memory. "Recent enough" predicates allow a program to control the circumstances under which it will (need to) obtain updates to a segment. Views allow the program to specify the *portions* of the segment for which it will obtain those updates. Extensions to the InterWeave API allow views to be changed by applications dynamically. Sharers of a single segment can have different views. So long as a program specifies its view correctly, it retains all of the advantages of InterWeave: a shared memory programming model, true

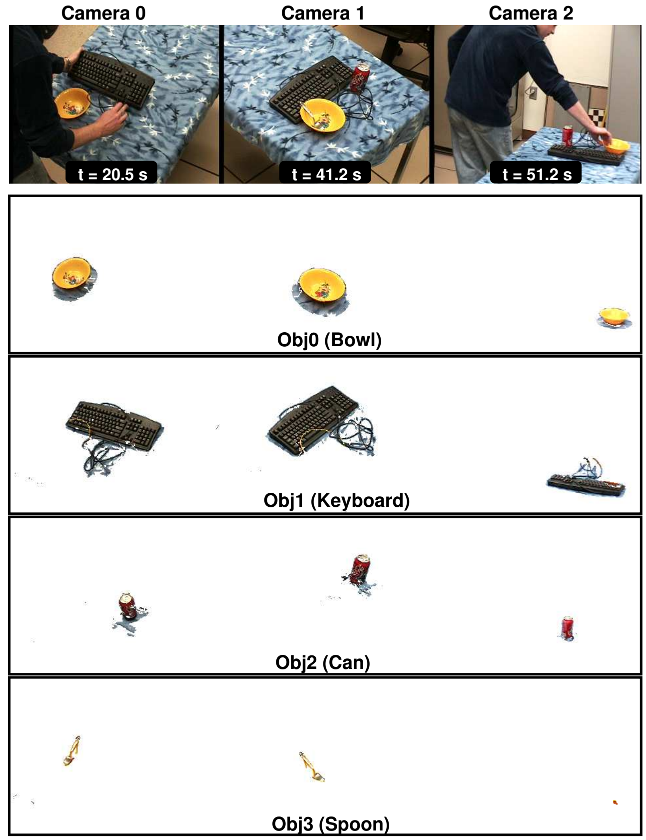


Figure 1: Three cameras monitor the same environment and cooperatively discover four objects by sharing relevant portions of their image cubes.

reference parameters for RPC, relaxed coherence models (a specification of "recent enough"), two-way diffing to identify, transmit, and update only the data that have changed, and transparent support for heterogeneous machines and languages. Our experience with dynamic views and relaxed coherence models on a variety of InterWeave applications indicates that programmers can provide high-level coherence information with very modest effort, and that InterWeave can in turn exploit this information to improve performance by as much as an order of magnitude.

Previous papers have discussed multi-level sharing [12] and heterogeneity [41] in InterWeave. In the current work we focus on exploiting application-level coherence information to optimize performance. We begin in Section 2 with a review of the InterWeave programming model and its implementation. We then describe the design and implementation of dynamic views in Section 3. In Section 4 we present performance results for three realistic applications: Internet proxy caching, intelligent environments, and interactive datamining. We discuss related work in more detail in Section 5, and conclude in Section 6.

2. INTERWEAVE BACKGROUND

As a prelude to the description of views in Section 3 and to the evaluation of both relaxed coherence and views in Section 4, we briefly review the design and implementation of InterWeave. A more detailed description can be found in other papers [12, 41].

2.1 Design of InterWeave

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. In keeping with wide-area distribution, InterWeave allows processes 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. In C, operations on shared data, including pointers, take precisely the same form as operations on non-shared data.

Servers may be replicated to improve availability and reliability. Data coherence among replicated servers is based on group communication [25]. A detailed discussion of server replication is beyond the scope of this paper.

2.1.1 Shared Data Allocation and Access

The unit of sharing in InterWeave is a self-descriptive data *segment* (a heap) 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 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. The call returns an opaque *handle* 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 = IW_malloc(h, my_type_desc);
*p = ...
IW_wl_release(h);
```

InterWeave 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.

As in multi-language RPC systems, the types of shared data in InterWeave must be declared in an Interface Description Language (IDL—currently Sun XDR). 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, and that allow the InterWeave library to translate data from one form to another when communicating between machines with different byte order, word length, alignment, or data representation.

InterWeave automatically translates and swizzles pointers inside shared data segments. When necessary (e.g. to obtain an initial pointer to a data structure using a MIP contained in a command-line parameter), a process can also translate explicitly between MIPs and local pointers using `IW_mip_to_ptr()` or `IW_ptr_to_mip()`.

2.1.2 Coherence

As clients modify an InterWeave segment, the changes are captured in a series of internally consistent *versions* of the segment. When a process first locks a shared segment (for either read or write access), 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" to use. If not, it obtains an update from the server. Twin and diff operations [8], extended to accommodate heterogeneous data formats, allow the implementation to perform an update (or to deliver changes to the server at the time of a write lock release) in time proportional to the fraction of the data that have changed.

InterWeave currently supports six different definitions of "recent enough". It is also designed in such a way that additional definitions (coherence models) can easily be added. Among the current models, *Full* coherence (the default) 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 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.

2.2 Implementation of InterWeave

InterWeave currently consists of approximately 31,000 lines of heavily commented C++ code. Both the client library and the server have been ported to a variety of architectures (Alpha, Sparc, x86, MIPS, and Power4), operating systems (Windows NT/2000/XP, Linux, Solaris, Tru64 Unix, IRIX, and AIX), and languages (C, C++, Fortran 77/90, and Java).

Our experiences with InterWeave have shown that it is scalable with respect to the number of clients [12] and that its performance is comparable to that of RPC parameter passing when transmitting previously uncached data [41]. When updating previously cached data, InterWeave's use of platform-independent diffs allows it to significantly outperform the straightforward use of RPC.

2.2.1 Client Library Implementation

When a process acquires a writer lock on a given segment, the InterWeave library asks the operating system to disable write access to the pages that comprise the local copy of the segment. When a write fault occurs, the `SIGSEGV` signal handler, installed by the InterWeave library at program startup time, creates a pristine copy, or *twin* [8], 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 writer lock, the library gathers local changes and converts them into machine-independent wire format in a process called *diff collection*. It then sends this diff back to the server. When a client acquires a reader lock and determines that its local cached copy of the segment is not recent enough to use under the desired coherence model, the client asks the server to build a diff that describes the data that have changed between the current local copy at the client and the master copy at the server. When the diff arrives, the library uses it to update the local copy in a process called *diff application*.

2.2.2 Server Implementation

Each server maintains an up-to-date copy of each of the segments for which it is responsible, and controls access to those segments.

To avoid an extra level of translation, the server stores both data and type descriptors in wire format. For each segment, the 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 and 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.

Upon receiving a diff from a client, an InterWeave server uses the diff to update its master copy. It also updates the version numbers associated with blocks and subblocks affected by the diff. At the time of a lock acquire, if the client's cached copy is not recent enough to use, 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 by comparing their version numbers with the client-presented version number, constructs a wire-format diff, and returns it to the client.

Supporting relaxed coherence models, in particular, delta coherence, is relatively easy with the help of the segment version numbers maintained by both the client and the server—it involves a simple comparison of version numbers at the server. Temporal coherence is almost as easy: clients support it by maintaining a real-time stamp for each locally cached segment and request an update when the difference between current time and the time stamp exceeds the coherence parameter. Diff coherence, by contrast, is a bit more complicated. For any client using Diff coherence, the server must be able to track the percentage of the segment that has been modified since the last version sent to the client. To minimize the cost of this tracking, the server conservatively assumes that all updates in each version are to independent portions of the segment. Thus, it suffices to keep track of the size of the diff for each version. By adding up the sizes for all versions newer than the last version seen by the client, the server can determine whether the client's diff coherence parameter has been exceeded and whether an update is necessary.

A variety of optimizations improve performance in common cases. A client that has the only cached copy of a segment will enter *exclusive mode*, in which it can acquire and release locks (both read and write) an arbitrary number of times, with no communication with the server. A segment that is usually modified in its entirety will enter *no-diff* mode, eliminating the need for write faults, twins, and diffs. Depending on the frequency of updates, a client/server pair will choose dynamically between *polling mode*, in which the client queries the server when it needs to evaluate its "recent enough" predicate, and *notification mode*, in which the server pushes this data to the client proactively. The utility of the *notification mode* is also a function of the number of clients, since the server needs to keep track of per-client coherence requirements. These and other optimizations are documented elsewhere [12, 41].

3. INTERWEAVE VIEWS

Views allow an InterWeave client to specify the portion of a segment in which it is currently interested, thereby relieving the underlying system from the need to maintain coherence for the rest of the segment. A view is constructed dynamically and can evolve over time. Sharers of a single segment can have different views. Where coherence models (Full, Temporal, Delta, etc.) address the temporal dimension of application-level coherence information, views address the spatial dimension. As with relaxed coherence models, it is the programmer's responsibility to define views correctly and to touch only the data covered by the current view.

3.1 InterWeave View Design

Each InterWeave view is explicitly associated with a segment and may contain an arbitrary number of *view units*. Each view unit is a contiguous portion of a block. The view may be specified by a pair of MIPs that refer to its start and end, respectively, or by equivalent local pointers, if the segment is locally cached.

A process can create an empty view given a segment handle:

```
IW_view_t v = IW_create_view(h);
```

Once a view has been created the process that created it can augment the view by attaching more view units:

```
IW_mip_t start, end;
bool recursive;
IW_attach_view_unit(v, start, end, recursive);
```

It can also correspondingly detach view units using `IW_detach_view_unit()`.

The `recursive` parameter indicates whether or not the view unit should be recursively expanded. If the `recursive` flag is set, the runtime includes in the view all other data that are reachable by following intra-segment pointers that originate inside the view unit. (Data reachable by following a pointer chain out of the segment and back in again will not be included.) Each contiguous region of additional data becomes an additional (implicitly specified) view unit in the view.

A view unit can be part of an array or multiple contiguous fields of a structure. We provide an API for convenient creation of frequently used view structures, such as slices of multi-dimensional arrays. Such structures comprise multiple view units, and can be attached to a view as a group. Recursive views are especially convenient for pointer-rich dynamic data structures, such as the subtree rooted at a given node or a linked list starting from a header node.

After view units have been attached to a view, the view can be activated for use by a call to `IW_activate_view()`, which transmits the view definition to the InterWeave server. At any given time, a single process can have at most one active view on a given segment. Once a process activates a view, future lock acquisitions will maintain coherence only for the portion of the segment covered by the view. As pointers inside a recursive view change, the view will be updated automatically by the runtime. New view units, whether recursively reachable or explicitly attached, will become effective (i.e., actually cached) at the time of the next lock acquire. A view remains in effect until it is disabled using `IW_deactivate_view()`. A view that is no longer needed can be destroyed using `IW_delete_view()`.

3.2 InterWeave View Implementation

The current InterWeave implementation is highly optimized for efficient data sharing in heterogeneous environments [41]. It employs sophisticated data structures to manage segment memory on both clients and servers, to track versions of segments, blocks, and subblocks, to swizzle pointers, and to generate wire-format diffs. In adding views to the system, we have tried to minimize the impact on performance when views are not being used. Only one minor change was required to the wire format itself (see Section 3.2.3).

The view implementation adds 2,500 lines of code to InterWeave. This code serves to maintain view descriptions at both client and server, and to generate view-specific diffs when a client must be updated. We elaborate on these points in the following subsections.

3.2.1 Client Side View Management

Within the client library, a view is represented by a hash table indexed by block serial number. Each entry in the table contains a list of the view units contained in a given block, together with their primitive offset ranges. When `IW_attach_view_unit()` is invoked, the runtime translates the `start` and `end` pointers of the view unit into a block serial number and a primitive offset range within the block, and updates the view's hash table accordingly.

To track updates to views, both the view itself and each of its view entries has a version number. When the process calls `IW_activate_view()` and then attempts to lock the segment, the runtime passes the hash table entries to the server. If view units are subsequently added to or deleted from the view, the client passes a list of these changes to the server the next time it acquires the lock.

So long as a client process keeps its promise to touch only the data covered by its view, the existing modification detection and wire format translation routines in InterWeave correctly collect any changes made by the client, and pass them to the server.

3.2.2 Server Side View Management

When a server receives a view definition from a client, it creates its own hash table, indexed by block serial number, to store the list of view units. For each large block in the table it also stores a bit vector indicating which subblocks are in the view.

For recursive view units, the server traverses the segment metadata to determine the full extent (*scope*) of the view. The traversal is driven by the type descriptors already maintained for the segment. For each view unit encountered, the server searches the type descriptor of the block to find the locations of pointers. It then builds a new view unit for each (strongly typed) pointed-to datum (not the entire block containing the datum) and adds these view units to the view. The traversal procedure stops when no more view units can be added.

When a client informs the server that it has added or deleted view units, the server updates its description of the view accordingly. When a block is deleted from a segment (by any client), the server automatically removes any view units contained in that block from all known client views. (The server also informs each client of the deleted block as part of the normal update mechanism when it next updates the client's cached version of the segment.)

With recursive view units, the scope of a view can change dynamically as pointers are reassigned. Before sending diffs to a client, any view with recursive view units needs (at least conceptually) to be re-expanded by recursively following pointers. To avoid this expensive operation, the server actually updates view scopes lazily and conservatively. Assisted by block version numbers, the server searches only subblocks that have changed since the last update sent to the client. For each pointer in such subblock, the server adds the pointed-to view unit into the view if it is not present yet. To avoid accidentally dropping useful view units, the server conservatively keeps the old pointed-to view unit in the view. With this strategy, a client may receive some view units that should already have been dropped. Although this is harmless semantically, it may waste bandwidth. As a trade-off, we set a threshold for the number of changed versions. When the threshold is exceeded, the server traverses all views and re-builds the view scopes.

3.2.3 Server Side Diff Collection

A server keeps the most recent version of the segments for which it is responsible. 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. Without views, the server can compute diffs for a client using the version number of the segment cached at the client and the version number associated with the server's master copy. The server simply updates the client with blocks or subblocks whose version is larger than the client's version. When views are in use, the process is similar except that the server now must consider the blocks or subblocks covered by the views and the version of these blocks or subblocks cached at the client. In this case, a single version number for each client no longer suffices.

Consider a client that activates a view of some segment S at version V_1 and obtains its first update at version V_2. At this point the data in the view have been updated to version V_2, but the data outside the view have not. Now suppose the client advances to a new version V_3, and then decides to add a new view unit v into the view (v might also be added by the runtime automatically as a result of pointer changes in a recursive view). Since the data d in v were not in the view before, the client-cached copy of d is at version V_1. If the server only maintains a single version number for the view, it will be unaware that d missed the updates between version V_1 and V_2. Simply put, while the server will know that a newly added view unit needs to be brought up to date, it won't know how out-of-date that view unit was before. Similar problems exist when dropped view units are added back into a view.

To address these problems, the server maintains some additional information for each client that has activated a view: (a) a *pre-view version*—the segment version number when the view was activated; (b) a *view version*—the segment version number when the client was most recently updated; and (c) a *view version table*—a hash table that tracks the version of each client-cached view unit, even if the view unit has been detached.

To collect a diff for a view unit covered by the current active view, the server must know which version of the view unit is cached by the client. For views whose scope has not changed since the last update to the client, the server knows that the client-cached view units have been updated to *view version*. This is the common case, and is handled efficiently by InterWeave. For view units added to the view since the last update, there are two cases. In one case, the corresponding view units are found in the *view version table*, so the server knows exactly which version is cached by the client. In the other case, the server can infer that the client's copy must be the *pre-view version*. Once the server determines a version for the client-cached copy, it computes the difference between the client's version and the current version, using the normal diff collection process already implemented in InterWeave. The server sends the diff to the client and updates the *view version table* to reflect the new version cached at the client. Figure 2 gives an example of the process described above.

While data not in the current view are not updated when acquiring a lock, the server must still inform the client of any changes to segment metadata, including: (a) added or deleted type descriptors; (b) the serial number of added or deleted blocks; and (c) the serial numbers of type descriptors of added blocks. This information is needed for management of segment memory space, and for swizzling pointers in a view that points to data outside the view. In the original InterWeave implementation, the data and metadata of a newly created block were sent to the client together. The ability to send the metadata by itself was the only change to InterWeave's wire format required to accommodate views.

4. PERFORMANCE AND APPLICATION EXPERIENCES

In this section, we evaluate the performance of InterWeave views using microbenchmarks and describe our experiences in specifying and exploiting high-level coherence information in three distributed applications. Unless otherwise specified, in all experiments, the

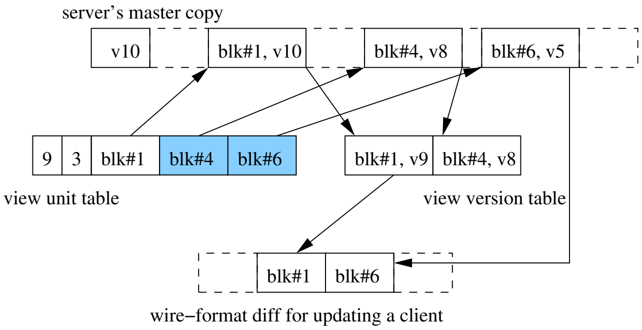


Figure 2: An InterWeave server collects a diff to update a client with an active view. Here `blk#4` and `blk#6` were added to the view after the last update to the client. `blk#4` was once in the view but was dropped after the client updated its version to `v8`. Since then `blk#4` has not been modified. `blk#6` was not in the view before. Using its *view version*, *pre-view version*, and *view version table*, the server constructs a diff containing the necessary updates for `blk#1` and `blk#6`, but not `blk#4`.

InterWeave server and client processes run on separate machines, each with a 2GHz Pentium IV processor running Linux 2.4.9, and are connected with either 100Mbps or 10Mbps Ethernet.

4.1 Microbenchmark Performance

With appropriate use of views, an application can help the InterWeave system reduce the overhead of coherence maintenance. This is especially beneficial for processes distributed over the Internet where bandwidth is limited. However, such benefits come at the cost of more metadata and maintenance operations on the server. In this section, we use microbenchmarks to evaluate the potential benefits of views and the overhead associated with them.

4.1.1 Non-recursive Views

The first experiment compares the time required for an InterWeave client to receive updates with or without non-recursive views. We arrange for two InterWeave client processes to share a segment consisting of 1000 blocks. Each block is a 160-element integer array. One process functions as a writer and updates every integer in the segment. The second process functions as a reader and uses *Full* coherence to read the segment after each update. We measure the latency for the reader process to acquire a reader lock. The latency is broken down into the communication time to transmit the client request and server update, the diff construction time on the server, and the translation time to apply the diff on the client. To factor in the network bandwidth, we experiment with two different connections between the the reader process and the InterWeave server, i.e. 100Mbps or 10Mbps Ethernet.

The results are shown in Figure 3. As can be seen from the figure, the communication time is significantly reduced due to the reduction in traffic by using views. The absolute reduction is more dramatic with the 10Mbps network (note the different scales on the two y axes). "Subblk. View" has slightly higher computation and communication overhead than "Blk. View" for both the client and server. On the server, when an entire block is in a view, the server only needs to collect changes in the block. When only a portion of a block is in the view, the server has to perform extra work to locate the portion of the block that is covered by the view. The resulting diff is also larger because more blocks, and thus more metadata, are in the diff. The larger diffs, increased metadata, and scattered changes increase the client's translation cost correspondingly.

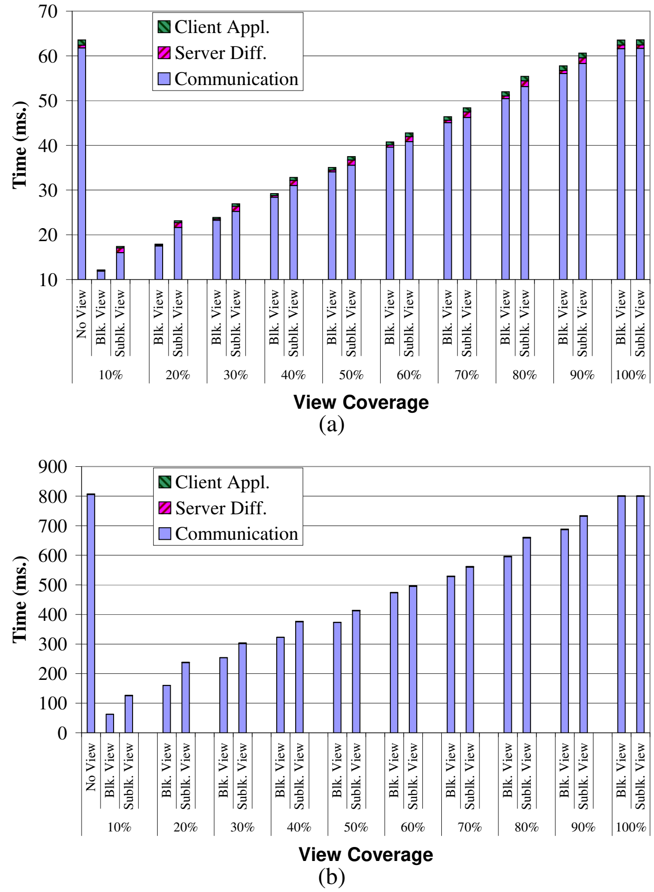


Figure 3: The effect of using non-recursive views. (a) 100Mbps network. (b) 10Mbps network. The x axis shows the increasing coverage of the view from 10% to 100%. The left-most bar in each graph is the baseline performance when no view is used. For cases involving views, we vary the scope of the view from a coverage of 10% to 100% of the segment. For each view coverage, the left bar ("Blk. View") uses a view consisting of x% of all blocks; the right bar ("Subblk. View") includes x% of each block in the view.

We plot the communication traffic (bytes transferred) for the "Blk. View" and "Subblk. View" in Figure 4. The bandwidth consumption is directly proportional to the percentage of total data contained in the current view.

4.1.2 Recursive Views

Our second experiment measures the time required for the server to maintain views with recursive view units. We arrange for one writer and one reader to share a segment that contains 50 doubly linked lists with header nodes. Besides pointers to the previous and next items in the list, each item contains 16 integers as a payload. We start with each list containing 1,000 items. The writer inserts 100 items at the end of each list and the reader locks the segment to get the updates. The reader includes a list into its view by adding the header of the list as a recursive view unit. Thus the items inserted into those lists will be automatically added to the view by InterWeave. We vary the coverage of views from 10% to all of

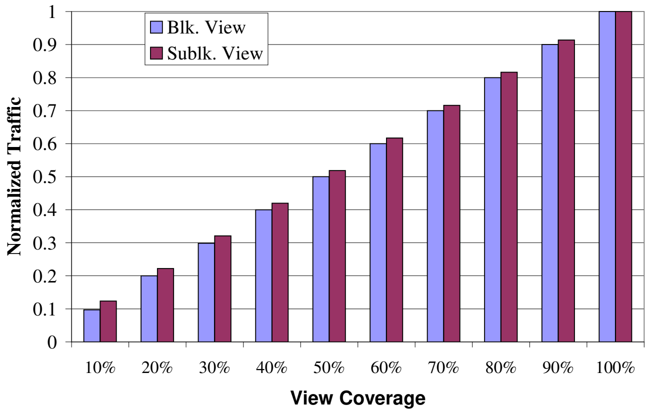


Figure 4: Communication traffic for "Blk. View" and "Subblk. View", normalized to the communication traffic without views.

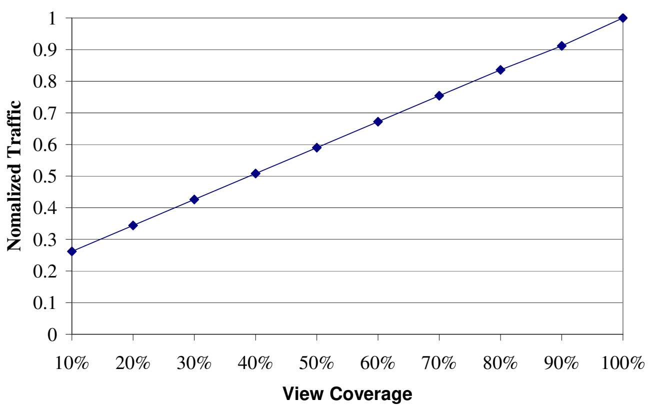
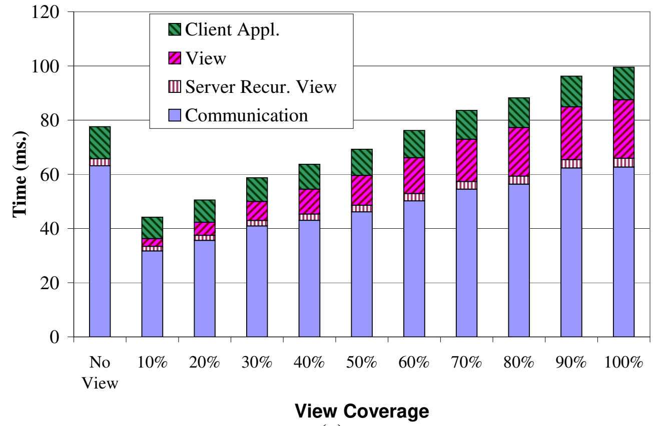


Figure 5: Communication traffic with recursive views, normalized to the communication traffic without views.



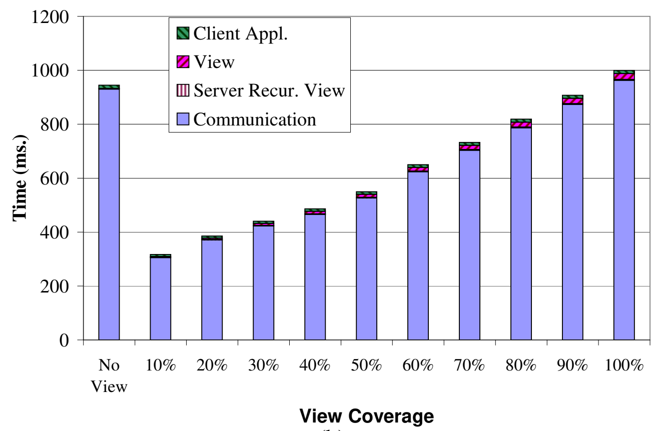


Figure 6: The effect of using recursive views. (a) 100Mbps network. (b) 10Mbps network. The left-most bar in each graph is the baseline performance without views.

the 50 lists. Again, we conduct experiments on both 100Mbps and 10Mbps Ethernet.

Figure 5 shows the reduction in communication traffic with recursive views. As described in Section 3.2.3, the InterWeave server always updates the client with full segment metadata. Because some of this metadata corresponds to data not included in the view, communication traffic is higher than in Figure 4.

In Figure 6, we compare the latency of client updates with and without views. The latency breakdown in this figure includes a new item, `Server Recur. View`, which is the cost of recursively computing the view scope on the server.

Unsurprisingly, there is a higher overhead associated with recursive views. This cost grows linearly as the view coverage increases. Combined together, the lower bandwidth reduction (see Figure 5) and the larger overhead of view maintenance actually cause the the performance of view coverage over 60% to become worse than that without views for the fast 100Mbps network. However, with a slower 10Mbps network, using views continues to be beneficial until it reaches 100% coverage. We expect Internet applications to benefit from views in InterWeave in most cases since the network conditions (e.g., bandwidth and latency) in the Internet are typically worse than that of a congestion-free 10Mbps local-area network. We are also investigating ways to maintain recursive views more efficiently.

4.2 Views in an Intelligent Environment Application

In Section 1, we described a distributed object discovery application in an intelligent environment [34, 35], where multiple nodes share their "image cubes". Each image cube is an array of images captured recently by the node's camera. As explained earlier, different nodes at different times access different portions of the cubes. Straightforward sharing of these cubes using InterWeave segments would be very inefficient, wasting large amounts of communication bandwidth if a node only wants to access a small portion of the cube. Similarly, splitting up the cube into multiple segments is cumbersome and difficult, especially given that the portion of the cube accessed by any node changes over time. In the following, we describe how we solve the problem using InterWeave views.

Since a full fledged intelligent environment is still under development, we use an application kernel to evaluate InterWeave. As shown in Figure 7, each camera node collects an image cube and stores it in an InterWeave segment shared by other nodes. Each image is stored as a separate block. An application running on a remote machine samples the images and executes a series of operations to find and analyze events in the cube. It first looks for events in the cube starting from the first image and coarsely sampling images through time. Once evidence of an event is detected

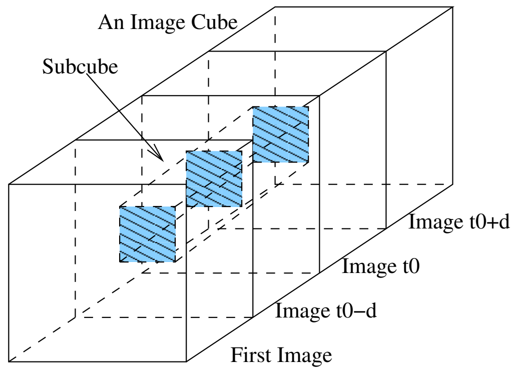


Figure 7: Finding and interpreting events in a shared image cube. The application samples images in the shared cube to detect interesting events. Later it examines the subcube containing that event.

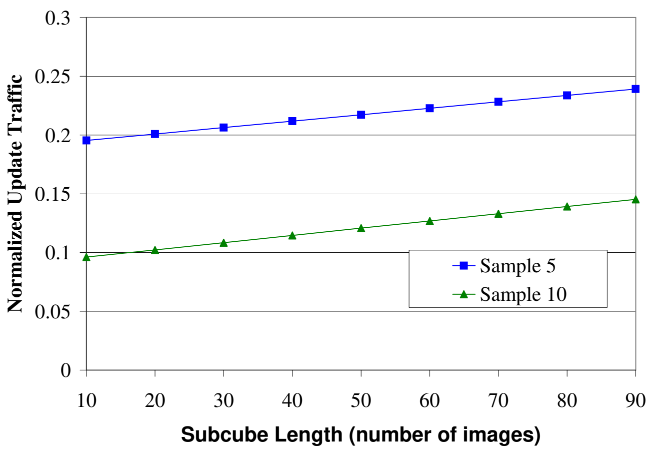


Figure 8: Bandwidth consumption with views under different sampling intervals, normalized to the bandwidth consumption without views. The x axis indicates the number of images intersected by the subcube.

in a sample image t_0, the application locates a minimal rectangular region that contains the event. To verify and interpret the event, it accesses the same region within d time steps before and after the event—the subcube from $t_0 - d$ to $t_0 + (d - 1)$.

Without using views, whenever a remote image is accessed, the entire image must be brought in, even though only a small part of it will ever be examined. Our solution is to use views to specify the portion of the image that will be scanned for events. Once an event is located within a sample image, we add the surrounding subcube to the view to more closely examine the evidence for the event.

Figure 8 shows the reduction in communication traffic achieved by using views. In this experiment, the image cube contains 100 recently collected images, each of size 320×240 pixels. The cube is sampled at two different rates, every 5th image ("Sample 5") or every 10th image ("Sample 10"). We assume the the discovery of an event is in the middle of the sampling process (i.e. at the 50th of 100 images in the cube). The application then examines the remote subcube centered around the event point. The cross section area of the subcube is 80×60 pixels and the length of the subcube varies from 10 to 90 pixels as indicated by the x axis in the figure.

As we can see from this figure, the communication traffic is significantly reduced. While the traffic increases as the length of the subcube increases for both "Sample 10" and "Sample 5", a closer comparison reveals that the former increases slightly faster than the latter. This is because the smaller sampling interval causes more images to be accessed and hence cached by the application. When an event is detected, the application needs a smaller amount of update data to construct the subcube containing the event.

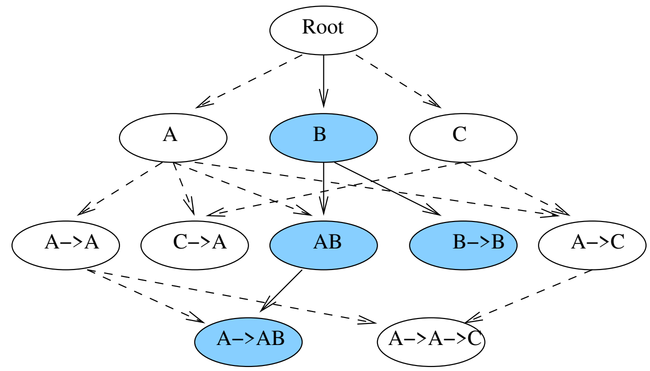


Figure 9: *Including queries* on a summary structure. Each node represents a meaningful dataming sequence. Each node has pointers to all other nodes for which it is a subsequence. Here the result for an including query concerning item B can be obtained by traversing the substructure rooted at node B.

4.3 Recursive Views in an Interactive and Incremental Datamining Application

We demonstrate the benefit of recursive views using an interactive and incremental datamining application. This application performs incremental sequence mining on a remote 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 one another in time. The goal is to find ordered sequences of items that are commonly purchased by individual customers over time.

In our experiments, both the database server and the datamining client are InterWeave clients. The database server reads from an active, growing database, and builds a summary data structure (a *lattice* of item sequences) that is used to answer mining queries, as shown in Figure 9. Each node in the lattice represents a potentially meaningful sequence of transactions s and contains pointers to other sequences containing s. The summary structure is shared between the database server and the mining client in a single InterWeave segment.

The mining client executes an *including query* over the summary structure, returning all sequences containing the query items. For example, in the example shown in Figure 9, an including query concerning item B will return the nodes highlighted in the figure (i.e., B, AB, $B{\rightarrow}B$ and $A{\rightarrow}AB$). Because each sequence node contains pointers to every node for which it is a subsequence, we can process an including query by starting from the nodes that have items that are required and then traversing the descendants of those nodes. If a client process is only interested in including queries on a certain set of items, it can save communication bandwidth by updating only the substructure rooted at those items.

Our sample database is generated by tools from IBM research [38]. It includes 100,000 customers and 1000 different items, with an average of 10 transactions per customer and a total of 5000 item sequence patterns of average length 4. The average transaction size is 2.5. The total database size is 20MB.

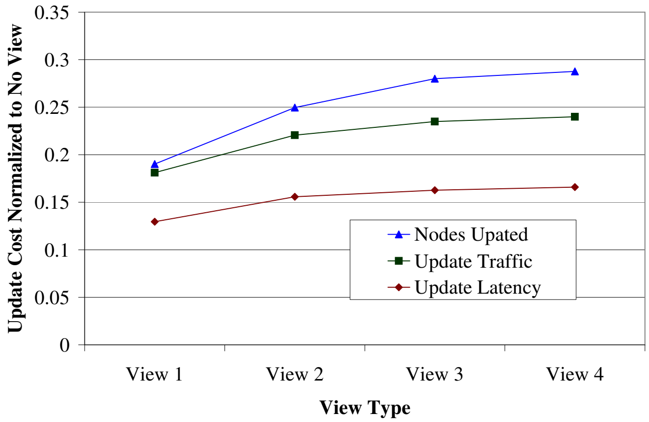


Figure 10: Update latency with views normalized to the cost without views in the datamining application.

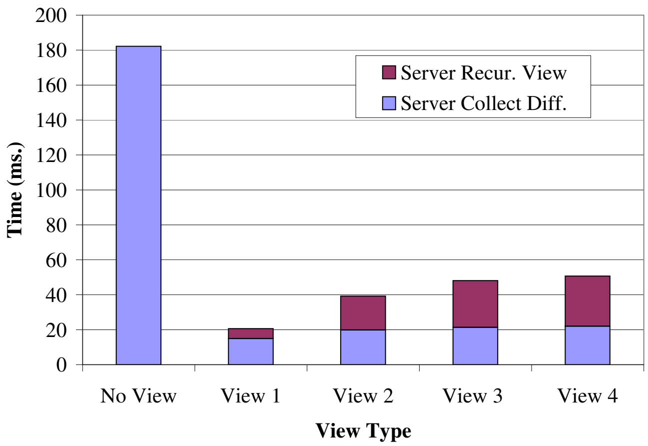


Figure 11: Server side overhead for constructing wire-format diffs for updates in the datamining application.

The summary structure is initially generated using half the database. The server then repeatedly updates the structure using an additional 1% of the database at each update. As a result, the number of nodes in the summary structure slowly and monotonically increases over time. For our tests we selected four items for which including queries would produce a relatively large number of sequences from the lattice.

Figure 10 compares the average update latency and bandwidth consumption seen at the client side using different view configurations. "View X" means X of our four selected items are added as recursive view units to the view. When the client acquires a lock on the segment, only the substructures rooted at those items are updated. Three different metrics are measured: update latency, update traffic, and the number of nodes updated. The figure demonstrates that recursive views tremendously reduce the update traffic and latency.

With fewer nodes to update, the server and the client also save diff processing time. Figure 11 shows the time spent constructing diffs and maintaining recursive views on the server. Recursive view maintenance overhead increases as the number of nodes added to the view increases. However, this overhead is adequately compensated for by the reduction in diff construction time, not to mention the reduced communication time due to lower traffic.

4.4 Scalable Sharing of Metadata in ICP

With the rapid growth of Internet traffic, hierarchical and cooperative proxy caching have proven to be effective in reducing bandwidth and improving client side access latency [15]. The Internet Caching Protocol (ICP), designed by the Harvard Harvest group [10], is perhaps the most popular sharing protocol, and is widely deployed. In ICP, cooperative proxies are organized into a hierarchical structure. Each proxy can have parents and siblings, all of which are referred to as peer proxies in this paper. When a proxy misses in its own local cache, it probes its peers for possible remote hits. If all of its peers miss, the proxy must go to the original web server on its own or ask its parent to fetch the object.

Unfortunately, ICP does not scale well as the size of caches or the number of cooperating proxies increases. On a local miss, the proxy sends query messages to its peers asking for the missed object. The number of these messages is quadratic in the number of peer proxies. To solve this problem, Summary Cache [15] and Cache Digest [33] have independently proposed similar solutions, in which each proxy keeps a compact directory of contents recently cached at every other proxy. Now when a proxy misses in its local cache, it consults its peer directories before sending out ICP query messages. Queries are sent only to those proxies indicated by the directory as having a recent version of the page (URL). Both solutions use *Bloom filters* [6] to represent the peer directories. A Bloom filter is a succinct randomized data structure that supports efficient membership queries.

There is a basic trade-off in the implementation of directories: frequent updates consume communication bandwidth, while outdated directories may introduce both false hits and false misses. False hits occur when the directory falsely indicates a URL exists in another proxy's cache. False misses occur when the directory does not list any proxy as containing the URL in its cache, even though the URL is actually cached at a peer. Use of a Bloom filter also entails a small but controllable false hit rate [6, 15].

Summary Cache [15] proposes a broadcasting update scheme. Each proxy broadcasts to its peers an update of its directory after a fixed percentage of changes have occurred (for example, broadcasting every 1% of its local changes). To reduce the size of a broadcast message, each time only the difference since the last update is broadcast. Cache Digest [33] uses an on-demand update scheme. Each proxy decides how often it needs an update from other proxies. A proxy can piggy-back update requests on query messages to its peers. Likewise, a proxy can inform its peers that a new directory update is available by piggy-backing the information on its responses to query messages.

Neither of the above schemes is ideal. Broadcasting requires that all peers be updated at a fixed rate. If one proxy requires more accurate information, each of its peers must also receive more frequent updates. More importantly, efficient and robust broadcasting support is usually not available in wide-area networks. While update on demand is more flexible, it consumes more bandwidth than broadcasting because it always transmits the entire directory (a Bloom filter) on each update. Since the Bloom filter is a randomized data structure, it is difficult to compress [30].

To evaluate the overhead incurred by updating peer directories on demand, we conduct an experiment on an ICP simulator, *proxycizer*, from the Crispy Squid Group at Duke University [14]. Since the original proxycizer implements neither Summary Cache nor Cache Digest, we augmented it with a directory implementation based on Cache Digest [33]. We use a trace file from the IRCache organization [21]. The trace records one day of requests to one of its proxies at Pittsburgh, Pennsylvania. The trace comprises 1.4 million HTTP requests with an average object size of 11.4KB. We

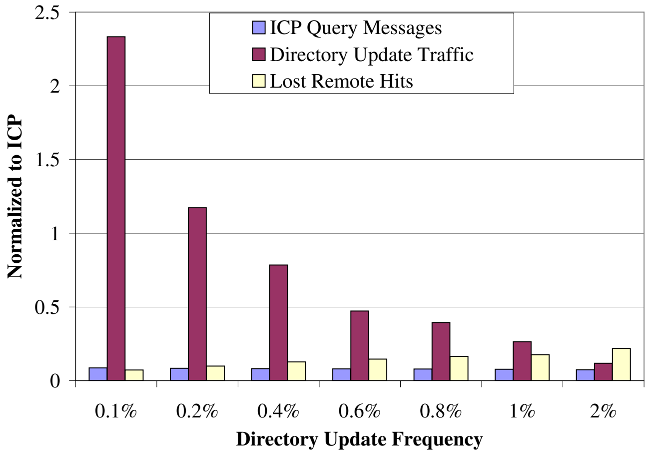


Figure 12: Using directories to reduce ICP query messages. The number of queries is normalized to that of ICP and the directory update traffic is normalized to the traffic of transmitting remote hit objects.

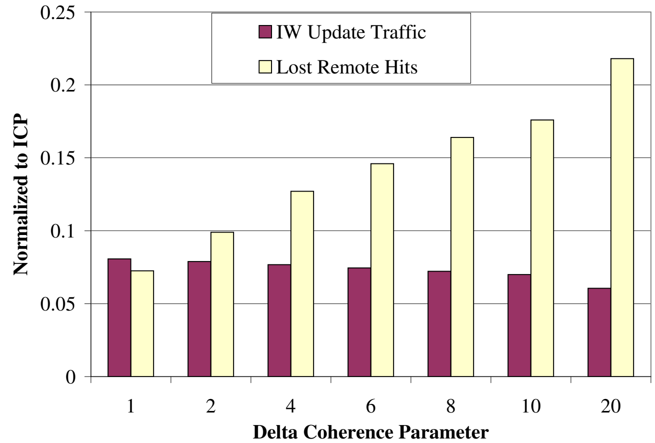


Figure 13: Using InterWeave segments to share peer directories. The x axis represents the parameter for the *Delta* coherence model. The directory update traffic is normalized to the traffic of transmitting remote hit objects. Note that the scale of the y axis is one order less than that in Figure 12.

first run the trace through proxycizer's ICP simulator with 4 simulated proxies, each with a disk cache of about 1.2GB (this allows space for approximately 100,000 objects). The trace is fed to the proxies in a round-robin manner. Table 1 summarizes the results.

Each proxy caches 3 Bloom filters to summarize the contents cached at each of its peer proxies. Each Bloom filter is 200KB long. (We use a Bloom filter longer than that specified in [33] in order to achieve a reasonable remote cache hit rate.) For each of its cached Bloom filters, a proxy requires an update once the proxy that is modeled by the Bloom filter has changed its content beyond a certain threshold. Figure 12 shows the number of query messages and the total communication traffic required to update the directories as we vary the update threshold from 0.1% to 2%. In this figure, we also show the percentage of remote hits lost due to the imprecise information in the directories.

The figure shows that using cached directories indeed significantly reduces the number of ICP query messages. However, the proxies voraciously consume bandwidth to update the directories. At the update threshold of 0.6%, the communication traffic to update the directories equals almost half the traffic for transmitting the objects themselves from hits in remote peers. The directory update traffic can be reduced by increasing the update threshold, with the trade-off of increasing lost remote hits.

We propose using InterWeave to automate the sharing of the directories among peer proxies. We can then reduce the traffic to update directories using InterWeave's relaxed coherence models and diffing. To evaluate this idea, we add an InterWeave simulator into the modified *proxycizer*. In the simulator, each proxy stores its own directory in a segment shared by other peer proxies. Each proxy updates its directory segment whenever 0.1% of its local disk cache changes. Other proxies access the directory by acquiring a reader lock on the segment.

Number of requests	1295000
Number of ICP query messages	2187207
Hits in local cache	28.4%
Hits in remote peer cache	9.7%
Aggregate size of remote hit objects	1860.89MB

Table 1: Simulation results for ICP.

We again run the previously described experiments, this time on the InterWeave-augmented simulator. To control the update frequency of cached segments of peer directories, we use the *Delta* coherence model with parameters x ranging from 1 to 20. These cause InterWeave to update the client's local segment cache whenever it is x versions older than the server's master version. Since each proxy updates its own directory segment when 0.1% of the cache changes (and thus creates a new version at that time), the parameters we used effectively correspond to the previously used update thresholds of 0.1% to 2%. The results are shown in Figure 13.

Comparing the InterWeave results with those in Figure 12, we see that InterWeave significantly reduces the communication traffic required to keep the peer directories up to date. Each time a proxy locks a peer directory segment for update, the segment server computes and transmits a diff capturing the difference between the directory's newest version and the proxy's cached version. Although Summary Cache [15] also broadcasts the differences between consecutive versions to reduce message size (which has not been compared here), InterWeave provides a much more flexible method for doing so without requiring complex coherence management code within the application. With InterWeave, the application (and in fact, each proxy) can change the update frequency simply by tuning the parameter for *Delta* coherence. In addition, cooperating peer proxies no longer have to communicate in lock step in order to coordinate with each other to update their directories.

In Figure 12, the choice of *Delta* coherence parameter exhibits a correlation with the rate of lost remote cache hits. Thus, each proxy can make individual decisions about how often it wants to get updates for each of its peers' directories, keeping the lost remote hit rate at a satisfactory level.

5. RELATED WORK

InterWeave finds context in an enormous body of related work. We focus here on some of the most relevant literature; additional discussion can be found in technical reports [11, 40].

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

[16], Legion [18], Globe [42], 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 [26] and CASCADE [13]) cache objects for locality of reference. While we speculate that relaxed coherence and views might be applicable to such systems, current implementations tend to rely on the inefficient retransmission of entire objects, or the transmission and replay of operation logs. 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 [45] 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. Precedents for the automatic management of pointers include Herlihy's thesis work [19], LOOM [24], and the more recent "pickling" (serialization) of Java [32].

Several "software-only" S-DSM systems have proposed that programmers explicitly identify the data to be modified in a critical section, either directly [20, 23] or by explicit [4, 36] or implicit [22] association with a synchronization object (lock). In contrast to systems that maintain coherence at the level of virtual memory pages, software-only S-DSM is less vulnerable to false sharing. In a similar vein, views in InterWeave relieve the system of the need to inform processes of updates to "uninteresting" portions of a data structure. In addition, InterWeave allows each process to customize its view as well as to change its coverage dynamically as needed.

Munin [8] is an early homogeneous S-DSM system that chooses among alternative coherence protocols (invalidate v. update, for example) based on program annotations that specify expected access patterns (migratory, widely shared, etc.). While it provides a release consistent programming model, Munin does not allow users to further relax coherence requirements with respect to reads and writes of the same location.

Stampede [31] is a system designed specifically with multimedia applications in mind. A data sharing abstraction called *space-time memory* 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 attempts to provide semantics similar to those of hardware shared memory, and therefore retains only the latest version of shared data.

Object View [28] uses programmer knowledge to classify objects according to their access patterns. Object views must be specified at compile time. InterWeave views do not rely on language extensions, and can be composed dynamically. Object Clusters [29] are closed sets of shared Java objects reachable and accessible only from a single root object by following object references. InterWeave similarly follows pointers to expand a recursive view scope, but has no restrictions on the choice of root objects.

Problem-Oriented Object Memory (POOM) [27] is an object model that allows exploitation of application specific semantics by relaxing strict consistency to achieve good performance for shared write-intensive data. Specifically, it allows multiple object replicas to be modified in parallel and uses a value "amalgamation" process to merge the state of diverged replicas of an object to a single meaningful value. In the POOM model the unit of consistency is still the entire object. InterWeave's relaxed coherence models serve mainly to improve performance for readers of shared data; coherence can be maintained for only part of a segment by using views.

Friedman [17] 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. Khazana [9] also proposes the use of multiple consistency models. The TACT system of Yu et al. [43] 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 semantics for writer locks.

6. CONCLUSION

InterWeave allows distributed applications to share strongly typed, pointer-rich data structures across heterogeneous hardware and software platforms. We described a new *dynamic view* mechanism for InterWeave, and discussed how views and relaxed coherence models exploit an application's high-level coherence requirements to optimize system performance. We demonstrated the convenience and effectiveness of these mechanisms with applications in intelligent environments, interactive datamining, and cooperative web proxy caching. We plan to evaluate and adopt techniques used in Peer-to-Peer (P2P) computing systems to improve InterWeave's scalability and fault tolerance, and to provide a shared state infrastructure for increasingly popular P2P applications. We are also considering transactional extensions to the InterWeave programming model, to enable processes to more easily modify a collection of segments consistently.

7. 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*, pages 101–110, 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. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[4] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the IEEE COMPCON '93*, pages 528–537, 1993.

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

[6] B. H. Bloom. Space/Time Trade-Off in Hash Coding with Allowable Errors. *Comm. of the ACM*, 13(7):422–426, July 1970.

[7] T. Brecht and H. Sandhu. The Region Trap Library: Handling Traps on Application-Defined Regions of Memory. In *Proc. of the 1999 USENIX Annual Technical Conf.*, pages 85–99, Monterey, CA, June 1999.

[8] 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.

[9] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *Proc. of the 18th Intl. Conf. on Distributed Computing Systems*, pages 562–571, Amsterdam, The Netherlands, May 1998.

[10] A. Chankhunthod, M. Schwartz, P. Danzig, K. Worrell, and C. Neerdaels. A Hierarchical Internet Object Cache. In *Proc. of the USENIX 1996 Technical Conf.*, pages 153–163, 1996.

[11] 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.

[12] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level Shared State for Distributed Systems. In *Proc. of the 2002 Intl. Conf. on Parallel Processing*, pages 131–140, Vancouver, BC, Canada, Aug. 2002.

[13] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proc., Middleware 2000*, pages 1–23, New York, NY, Apr. 2000.

[14] Duke University Computer Science Department. Proxycizer. http://www.cs.duke.edu/ari/cisi/Proxycizer/.

[15] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE Trans. on Networking*, 8(3):281–293, 2000.

[16] 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.

[17] R. Friedman. Implementing Hybrid Consistency with High-Level Synchronization Operations. In *Proc. of the 12th ACM Symp. on Principles of Distributed Computing*, pages 229–240, Ithaca, NY, Aug. 1993.

[18] 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*, pages 89–99, Aug. 1996.

[19] 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.

[20] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Trans. on Computer Systems*, 11(4):300–318, Nov. 1993.

[21] IRCache Project. Originally sponsored by the National Science Foundation (grants NCR-9616602 and NCR-9521745), and the National Laboratory for Applied Network Research. http://www.ircache.net.

[22] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge Between Release Consistency and Entry Consistency. In *Proc. of the 8th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 451–473, Padua, Italy, June 1996.

[23] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 213–228, Copper Mountain, CO, Dec. 1995.

[24] 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.

[25] B. Kemme and G. Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. In *Proc. of the 18th Intl. Conf. on Distributed Computing Systems*, pages 156–163, Amsterdam, The Netherlands, May 1998.

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

[27] A. Kristensen and C. Low. Problem-oriented Object Memory: Customizing Consistency. In *OOPSLA '95 Conf. Proc.*, pages 399–413, Austin, TX, Oct. 1995.

[28] L. Lipkind, I. Pechtchanski, and J. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *OOPSLA '99 Conf. Proc.*, pages 447–460, Denver, CO, Nov. 1999.

[29] J. Maassen, T. Kielmann, and H. E. Bal. Efficient Replicated Method Invocation in Java. In *Proc., Java Grande 2000*, pages 88–96.

[30] M. Mitzenmacher. Compressed Bloom Filters. In *Proc. of the 20th ACM Symp. on Principles of Distributed Computing*, pages 144–150, Aug. 2001.

[31] 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.

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

[33] A. Rousskov and D. Wessels. Cache Digests. *Computer Networks and ISDN Systems*, 30(22-23):2155–2168, 1998.

[34] B. C. S. Sanders, R. C. Nelson, and R. Sukthankar. A Theory of the Quasi-static World. In *Proc. of the 16th Intl. Conf. on Pattern Recognition*, pages 3:1–6, Quebec City, PQ, Canada, Aug. 2002.

[35] B. C. S. Sanders, R. C. Nelson, and R. Sukthankar. The OD Theory of TOD: The Use and Limitations of Temporal Information for Object Discovery. In *Proc. of the 18th Natl. Conf. on Artificial Intelligence*, pages 777–784, Edmonton, AB, Canada, July 2002.

[36] H. S. Sandhu, B. Gamsa, and S. Zhou. The Shared Regions Approach to Software Cache Coherence on Multiprocessors. In *Proc. of the 4th ACM Symp. on Principles and Practice of Parallel Programming*, pages 229–238, San Diego, CA, May 1993.

[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*, pages 211–220, 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. L. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 170–183, St. Malo, France, Oct. 1997.

[40] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Support for Machine and Language Heterogeneity in a Distributed Shared State System. TR 783, Computer Science Dept., Univ. of Rochester, May 2002.

[41] C. Tang, D. Chen, S. Dwarkadas, and M. L. Scott. Efficient Distributed Shared State for Heterogeneous Machine Architectures. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May 2003.

[42] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, 7(1):70–78, Jan.-Mar. 1999.

[43] 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*, pages 305–318, San Diego, CA, Oct. 2000.

[44] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, pages 29–42, Banff, Canada, Oct. 2001.

[45] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous Distributed Shared Memory. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):540–554, Sept. 1992.