Efficient Distributed Shared State for Heterogeneous Machine Architectures *

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

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

Abstract

InterWeave is a distributed middleware system that supports the sharing of strongly typed, pointer-rich data structures across heterogeneous platforms. As a complement to RPC-based systems such as CORBA, .NET, and Java RMI, InterWeave allows processes to access shared data using ordinary reads and writes. Experience indicates that InterWeave-style sharing facilitates the rapid development of distributed applications, and enhances performance through transparent caching of state.

In this paper, we focus on the aspects of InterWeave specifically designed to accommodate heterogeneous machine architectures. Beyond the traditional challenges of message-passing in heterogeneous systems, InterWeave (1) identifies and tracks data changes in the face of relaxed coherence models, (2) employs a wire format that captures not only data but also diffs *in a machine and language-independent form, and (3) swizzles pointers to maintain long-lived (cross-call) address transparency. To support these operations, InterWeave maintains an extensive set of metadata structures, and employs a variety of performance optimizations. Experimental results show that InterWeave achieves performance comparable to that of RPC parameter passing when transmitting previously uncached data. When updating data that have already been cached, InterWeave's use of platform-independent diffs allows it to significantly outperform the straightforward use of RPC.*

1. Introduction

With the rapid growth of the Internet, more and more applications are being developed for (or ported to) wide area networks in order to take advantage of resources available at distributed sites. Examples include e-commerce, computer-supported collaborative work, intelligent environments, interactive data mining, and remote scientific visualization. Conceptually, most of these applications involve some sort of *shared state*: information that is needed at more than one site, that has largely static structure (i.e., is not streaming data), but whose content changes over time.

For the sake of locality, "shared" state must generally be cached at each site, introducing the need to keep copies up-to-date in the face of distributed updates. Traditionally the update task has relied on ad-hoc, application-specific protocols built on top of RPC-based systems such as CORBA, .NET, and Java RMI. We propose instead that it be automated. Specifically, we present a system known as InterWeave that allows programs written in multiple languages to map shared *segments* into their address space, regardless of Internet address or machine type, and to access the data in those segments transparently and efficiently once mapped.

We do not propose in most cases to supplant RPC with a shared-memory style of programming. Rather, we take as given that many distributed applications will continue to be based on remote invocation. We see InterWeave-style shared state as a *complement* to RPC. In this role, InterWeave serves to (a) eliminate hand-written code to maintain the coherence and consistency of cached data; (b) support genuine reference parameters in RPC calls, eliminating the need to pass large structures repeatedly by value, or to recursively expand pointer-rich data structures using deep-copy parameter modes; and (c) reduce the number of "trivial" invocations used simply to put or get data.

Unfortunately, sharing is significantly more complex in a heterogeneous, wide-area network environment than it is in software distributed shared memory (S-DSM) systems such as TreadMarks [1] and Cashmere [14]. With rare exceptions, S-DSM systems assume that clients are part of a single program, written in a single language, running on identical hardware nodes on a system-area network. InterWeave must support coherent, persistent sharing among programs written in multiple languages, running on multiple machine types, spanning potentially very slow Internet links. Coherence and consistency (including persistence) [5], and support for Java [4] are the subjects of other papers; we concentrate here on the mechanisms required to accommodate machine heterogeneity.

*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; and by the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460.

RPC systems have of course accommodated multiple machine types for many, many years. They do so using stubs that convert value parameters to and from a machine-independent *wire format*. InterWeave, too, employs a wire format, but in a way that addresses three key challenges not found in RPC systems. First, to minimize communication bandwidth and to support relaxed coherence models (which allow cached copies of data to be slightly out of date), InterWeave must efficiently identify all changes to a segment, and track those changes over time. Second, in order to update cached copies, InterWeave must represent not only data, but also *diffs* (concise descriptions of only those data that have changed) in wire format. Third, to support linked structures and reference parameters, InterWeave must *swizzle* pointers [16] in a way that turns them into appropriate machine addresses. To support all these operations, InterWeave maintains metadata structures comparable to those of a sophisticated language reflection mechanism, and employs a variety of algorithmic and protocol optimizations specific to distributed shared state.

When translating and transmitting previously uncached data, InterWeave achieves throughput comparable to that of standard RPC packages, and 20 times faster than Java RMI [4]. When the data have been cached and only a fraction of them are changed, InterWeave's translation cost and bandwidth requirements scale down proportionally and automatically; pure RPC code requires ad-hoc recoding to achieve similar performance gains.

We describe the design of InterWeave in more detail in Section 2. We provide implementation details in Section 3, and 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 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. InterWeave servers are oblivious to the programming languages used by clients, and the client libraries may be different for different programming languages. Figure 1 presents client code for a simple shared linked list. The InterWeave API used in the example is explained in more detail in the following sections. For consistency with the example, we present the C version of the API. Similar versions exist for C++, Java, and Fortran.

2.1. Data Allocation

The unit of sharing in InterWeave is a self-descriptive *segment* (a heap) within which programs allocate strongly

```
node_t *head;        IW_handle_t h;
void list_init(void) {
  h = IW_open_segment("host/list");
  head = IW_mip_to_ptr("host/list#head");
}

node_t *list_search(int key) {
  IW_rl_acquire(h);      // read lock
  for (node_t *p=head->next; p; p=p->next)
    if(p->key==key) {
      IW_rl_relese(h);   // read unlock
      return p;
    }
  IW_rl_release(h);      // read unlock
  return NULL;
}

void list_insert(int key) {
  node_t *p;
  IW_wl_acquire(h);      // write lock
  p = (node_t*)IW_malloc(h, IW_node_t);
  p->key = key;
  p->next = head->next;
  head->next = p;
  IW_wl_release(h);      // write unlock
}
```

Figure 1. Shared linked list in InterWeave. Variable `head` **points to an unused header node; the first real item is in** `head->next`**.**

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, `IW_open_segment()` 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()`.

As in multi-language RPC systems, the types of shared data in InterWeave must be declared in an interface description language (IDL). The InterWeave IDL compiler translates these declarations into the appropriate programming language(s) (C, C++, Java, Fortran). It also creates initialized *type descriptors* that specify the layout of the types on the specified machine. The descriptors must be registered with the InterWeave library prior to being used, and are passed as the second argument ("IW_node_t" in Figure 1) in calls to `IW_malloc()`. These conventions allow the library to translate to and from wire format, ensuring that each type will have the appropriate machine-specific byte order, alignment, etc. in locally cached copies of segments.

Synchronization (to be discussed further in Section 2.2) takes the form of reader-writer locks that take a segment handle as parameter. A process must hold a writer lock on a segment in order to allocate, free, or modify blocks.

Given a pointer to a block in an InterWeave segment, or to data within such a block, a process can create a corresponding MIP: "`IW_mip_t m = IW_ptr_to_mip(p)`".

This MIP can then be passed to another process through a message, a file, or a parameter of a remote procedure. 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, and returns a local machine address. Actual data for the segment will not be copied into the local machine unless and until the segment is locked.

It should be emphasized that `IW_mip_to_ptr()` is primarily a bootstrapping mechanism. Once a process has one pointer into a data structure (e.g. the `head` pointer in our linked list example), any data reachable from that pointer can be directly accessed in the same way as local data, even if embedded pointers refer to data in other segments. InterWeave's pointer-swizzling and data-conversion mechanisms ensure that such pointers will be valid local machine addresses. It remains the programmer's responsibility to ensure that segments are accessed only under the protection of reader-writer locks.

2.2. Coherence

When modified by clients, InterWeave segments move over time through a series of internally consistent states. When a process first locks a shared segment (for either read or write), 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. An adaptive polling/notification protocol [5] often allows the client library to avoid communication with the server when updates are not required. Twin and diff operations [3], extended to accommodate heterogeneous data formats, allow the implementation to perform an update in time proportional to the fraction of the data that has changed.

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. To minimize the cost of segment updates, the server remembers, for each block, the version number of the segment in which that block was last modified. This information allows the server to avoid transmitting copies of blocks that have not changed. As partial protection against server failure, InterWeave periodically checkpoints segments and their metadata to persistent storage.

3. Implementation

In this section, we describe the implementation of the InterWeave server and client library. For both the client and the server we first describe the structure of the metadata, followed by the algorithms for modification tracking, wire-format diffing, and pointer swizzling.

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, and MIPS), operating systems (Windows NT/2000/XP, Linux, Solaris, Tru64 Unix, and IRIX), and languages (C, C++, Fortran, and Java).

3.1. Client Implementation

Memory management and metadata. As described in Section 2, InterWeave presents the programmer with two granularities of shared data: *segments* and *blocks*. Each block must have a well-defined type, but this type can be a recursively defined structure of arbitrary complexity, so blocks can be of arbitrary size. Every block has a serial number within its segment, assigned by `IW_malloc()`. It may also have an optional 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.

The copy of a segment cached by a given process need not be contiguous in the application's virtual address space, so long as individually `malloc`ed blocks are contiguous. The InterWeave library can therefore implement a segment as a collection of *subsegments*, invisible to the user. Each subsegment is contiguous, and can be any integral number of pages in length. These conventions support blocks of arbitrary size, and ensure that any given page contains data from only one segment. New subsegments can be allocated by the library dynamically, allowing a segment to expand over time.

An InterWeave client manages its own heap area, rather than relying on the standard C library function `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 to allow InterWeave to quickly locate blocks by name, serial number, or address.

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

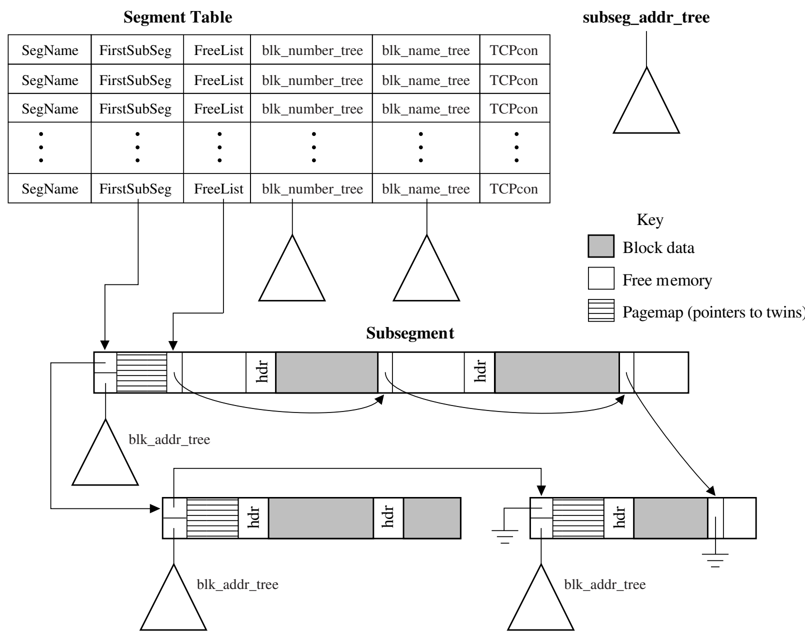

Figure 2. Simplified view of InterWeave client data structures: the segment table, subsegments, and blocks within segments. Type descriptors, pointers from balanced trees to blocks and subsegments, and footers of blocks and free space are not shown.

segment 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 (*blk_number_tree*), the other by block symbolic name (*blk_name_tree*); together they support translation from MIPs to local pointers. An additional global tree contains the subsegments of all segments, sorted by address (*subseg_addr_tree*), and each subsegment has a balanced tree of blocks sorted by address (*blk_addr_tree*); together these trees support modification detection and translation from local pointers to MIPs. Segment table entries may also include a cached TCP connection over which to reach the server. Free space within a segment is kept on a linked list, with a head pointer in the segment table.

Modification tracking. When a process acquires a write lock on a given segment, the InterWeave library asks the operating system to write protect the pages that compose 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* [3], of the page in which the write fault occurred. It saves a pointer to that twin in the faulting subsegment's header 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 page fault handler places a pointer to the twin in the ith entry of a structure called the *pagemap*, located in j's header (see Figure 2). With *subseg_addr_tree*, the handler can easily determine i and j. Together, the pagemaps and the linked list of subsegments in a given segment allow InterWeave to quickly find pages to be diffed when the coherence protocol needs to send an update to the server.

Diff creation and translation. When a process releases a write lock, the library gathers local changes and converts them into machine-independent wire format in a process called *diff collection*. Figure 3 shows an example of this process. The changes are expressed in terms of segments, blocks, and offsets of primitive data units (integers, doubles, chars, etc.), rather than pages and bytes. A wire-format block diff consists of a block serial number, the length of the diff (measured in bytes), and a series of run length encoded data changes, each of which consists of the starting point and length of the change (both measured in primitive data units), and the updated data (in wire format).

The diffing routine must have access to type descriptors in order to compensate for local byte order and alignment, and in order to swizzle pointers. The content of each descriptor specifies the substructure and layout of its type. For primitive types there is a single pre-defined descriptor. For derived types there is a descriptor indicating either an array,

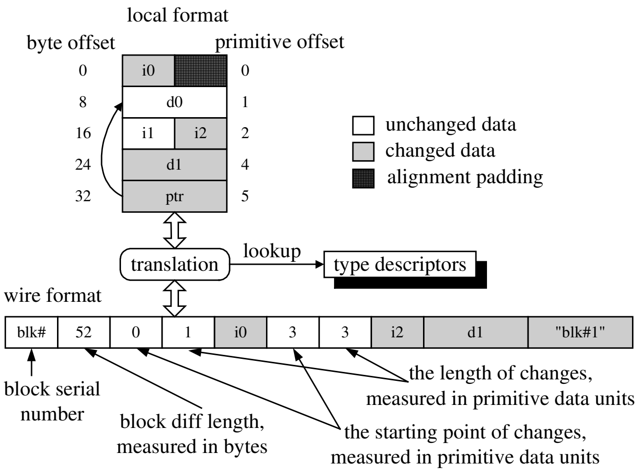

Figure 3. Wire format translation of a structure consisting of three integers (`i0`–`i2`), two doubles (`d0`–`d1`), and a pointer (`ptr`). All fields except `d0` and `i1` are changed.

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. For structures, the descriptor records both the *byte offset* of each field from the beginning of the structure in local format, and the machine-independent *primitive offset* of each field, measured in the number of primitive data units. Like blocks, type descriptors have segment-specific serial numbers to be used by the server and client in wire-format messages.

When translating local modifications into wire format, the diffing routine scans the list of subsegments of the segment and the pagemap within each subsegment. When it finds a modified page, it performs a word-by-word comparison of the current version of the page and the page's twin, identifying the first (*change_begin*) and last (*change_end*) words of a contiguous *run* of modified words. It searches the *blk_addr_tree* within the subsegment to identify the block that spans *change_begin*, and translates changes to this block into wire format.

To translate changes to a block into wire format, the diffing routine uses the type descriptor pointer stored in the header of each block to identify the primitive datum corresponding to *change_begin* and its primitive offset from the beginning of the block. Consecutive type descriptors, from *change_begin* to *change_end* (or the end of the block, whichever comes first), are then retrieved sequentially to convert the run into wire format. When done, the diffing routine translates the next block covered by current run if it is not finished yet, or returns to word-by-word comparison to find the next run to be translated.

When a client acquires a read 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 been 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*. In the inverse of diff collection, the diff application routine uses type descriptors to identify the local-format bytes that correspond to primitive data changes in the wire-format diff.

Pointer swizzling. To accommodate references, InterWeave relies on pointer swizzling [16]. To swizzle a local pointer to a MIP, the library first searches the *subseg_addr_tree* for the subsegment spanning the pointed-to address. It then searches the *blk_addr_tree* within the subsegment to find the pointed-to block. It subtracts the starting address of the block from the pointed-to address to obtain the byte offset. With the help of the type descriptor stored in the block header, the library then maps the byte offset into the primitive offset inside the block. Finally, the library converts the block serial number and primitive offset into strings, and concatenates them with the segment name to form a MIP. Swizzling a MIP into a local pointer is an analogous inverse process.

3.2. Server Implementation

Segment metadata. An InterWeave server can manage an arbitrary number of segments, and maintains an up-to-date copy of each of them. It also controls access to these segments. To avoid an extra level of translation, the server stores both data and type descriptors in wire format. It keeps track of segments, blocks, and *subblocks*.

An InterWeave server maintains an entry for each of its segments in a segment hash table keyed by the segment name. The blocks of a given segment are organized into a balanced tree sorted by their serial numbers (*svr_blk_number_tree*) and a linked list sorted by their version numbers (*blk_version_list*). The linked list is separated by markers into sublists, each of which contains blocks with the same version number. Markers are also organized into a balanced tree sorted by version number (*marker_version_tree*). Pointers to all these data structures are kept in the segment table, along with the segment name.

To track changes at a sufficiently fine grain, the server divides large blocks into smaller contiguous subblocks. It then stores version numbers for these subblocks in a per-block array. When a client needs an updated version of the segment, the server sends the (full content of the) subblocks that are newer than the version of the segment currently cached at the client. (Modified subblocks are interpreted by the client simply as runs of modified data; clients are unaware of subblocks.) In order to avoid unnecessary data relocation, MIPs and character string data are stored separately from their blocks, since they can be of variable size.

Unlike the InterWeave client library, which obtains its type descriptors from the application program, the InterWeave server must obtain its type descriptors from clients, and convert them to a form that describes the layout of blocks in machine-independent wire format.

Modification tracking and diff creation. Upon receiving a diff, an InterWeave server first appends a new marker to the end of the *blk_version_list* and inserts the marker into the *marker_version_tree*. Newly created blocks are then appended to the end of the list. Modified blocks are first located by searching the *svr_blk_number_tree*, and then are moved to the end of the list.

When an InterWeave client acquires a lock for a segment, the server and client library collaboratively decide whether the client needs to update the local copy of the segment, based on the coherence model requested by the client.

Among the relaxed coherence models currently supported by InterWeave [5], *Delta* coherence 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. The InterWeave library maintains a real-time stamp for each cached segment at the client to support Temporal coherence. With the segment version numbers maintained by both the client and server, supporting delta coherence is as simple as a comparison of version numbers.

Diff coherence, however, takes more effort. 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.

When an update to the client is necessary, the server traverses the *marker_version_tree* to locate the first marker whose version is newer than the client's version. As we described above, the *blk_version_list* is organized such that in this list all blocks after that marker have some subblocks that need to be sent to the client. Those modified subblocks are identified by version numbers associated with each subblock. The server then constructs a wire-format diff and sends it back to the client. Diff collection and application on the server are similar to their counterpart on the client, except that searches are guided by the server's machine-independent type descriptors, and the modifications are collected by inspecting version numbers of blocks and subblocks. Because MIPs are kept in wire format on the server, there is no need for the server to swizzle pointers.

3.3. Optimizations

Several optimizations improve the performance of InterWeave in important common cases. We describe here those related to memory management and heterogeneity.

Data layout for cache locality. InterWeave's knowledge of data types and formats allows it to organize blocks in memory for the sake of spatial locality. When a segment is cached at a client for the first time, blocks that have the same version number, meaning they were modified by another client in a single write critical section, are placed in contiguous locations, in the hope that they may be accessed or modified together by this client as well. Currently we do not relocate blocks in an already cached copy of a segment.

Diff caching. The server maintains a cache of diffs that it has received recently from clients, or collected recently itself, in response to client requests. These cached diffs can often be used to respond to future requests, avoiding redundant collection overhead. In most cases, a client sends the server a diff, and the server caches and forwards it in response to subsequent requests.

No-diff mode. As in TreadMarks [1], a client that repeatedly modifies most of the data in a segment (or a block within a segment) will switch to a mode in which it simply transmits the whole segment (or individual block) to the server at every write lock release. This *no-diff* mode eliminates the overhead of `mprotects`, page faults, and the creation of twins and diffs. Moreover, translating an entire block is more efficient than translating diffs. A client with a segment in no-diff mode will periodicallly switch back to diffing mode to capture changes in application behavior.

Isomorphic type descriptors. For a given data structure declaration in IDL, our compiler outputs a type descriptor most efficient for runtime translation rather than strictly following the original type declaration. For example, if a struct contains 10 consecutive integer fields, the compiler generates a descriptor containing a 10-element integer array instead. This altered type descriptor is used only by the InterWeave library, and is invisible to the programmer; the language-specific type declaration always follows the structure of the IDL declaration.

Diff run splicing. In a diffing operation, if one or two adjacent words are unchanged while both of their neighboring words are changed, we treat the entire sequence as changed in order to avoid starting a new run length encoding section in the diff. It already costs two words to specify a head and a length in the diff, and the spliced run is faster to apply. Splicing is particularly effective when translating double-word primitive data in which only one word has changed.

Last-block searches. On both the client and the server, block predictions are used to avoid searching the balanced tree of blocks sorted by serial number when mapping serial

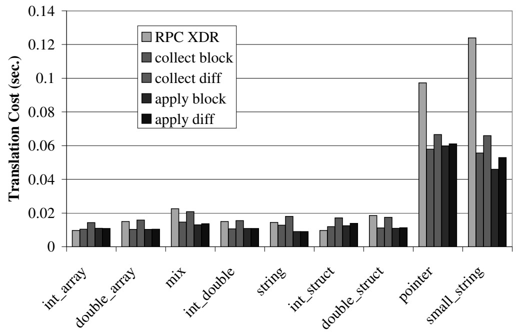

Figure 4. Client's cost to translate 1MB of data.

numbers in wire format to blocks. Based on the observation that blocks modified together in the past tend to be modified together in the future, we predict the next changed block in the diff to be the next consecutive block in memory for the client (since the layout is based on prior consecutive modification of the blocks), or the next block in the *blk_version_list* (which is sorted by version number) for the server.

4. Performance Results

Due to space limitations, we focus here on our evaluation of InterWeave's performance related to heterogeneity. More results can be found in the TR version of this paper [15]. Unless otherwise noted, the results were collected on a 500MHz Pentium III machine, with 256MB of memory, a 16KB L1 cache, and a 512KB L2 cache, running Linux 2.4.18.

4.1. Basic Translation Costs

Figure 4 shows the overhead to translate various data structures from local to wire format and vice versa on the client, assuming that all data has been modified and the entire structure is transmitted. In each case, we arrange for the total amount of data to equal 1MB; what differs is data formats and types. `Int_array` and `double_array` employ a large array of integers or doubles, respectively. `Int_struct` and `double_struct` employ an array of structures, each with 32 integer or double fields, respectively. `String` and `small_string` employ an array of strings, each of length 256 or 4 bytes, respectively. `Pointer` employs an array of pointers to integers. `Int_double` employs an array of structures containing integer and double fields, intended to mimic typical data structures in scientific programs. `Mix` employs an array of structures containing integer, double, string, small string, and pointer fields, intended to mimic typical data structures in non-scientific programs such as calendars and CSCW.

The "collect diff" and "apply diff" bars in Figure 4 show the overhead of translation to and from wire format, respectively, when diffing every block. The "collect block" and "apply block" bars show the corresponding overheads when diffing has been disabled (*no diff* mode). For comparison purposes we also show the overhead of translating the same data via RPC parameter marshaling functions generated with the standard Linux `rpcgen` tool. In our experiments, we found unmarshaling costs to be roughly identical. All optimizations described in Section 3.3 were enabled. All of them provided measurable improvements in performance and/or bandwidth; space constraints preclude a separate presentation here for all but no-diff mode.

Generally speaking, InterWeave overhead is comparable to that of RPC. Averaged across our 9 experiments, "collect block" and "apply block" are 25% faster than RPC; "collect diff" and "apply diff" are 8% faster. It is clear that `rpcgen` is not good at marshaling pointers and small strings. Excluding these two cases, InterWeave in *no diff* (collect/apply block) mode is still 18% faster than RPC. When diffing is performed, InterWeave is 0.5% slower than RPC. "Collect block" is 39% faster than "collect diff" on average, and "apply block" is 4% faster than "apply diff", justifying the use of the *no diff* mode.

When RPC marshals a pointer, deep copy semantics require that the pointed-to data, an integer in this experiment, be marshaled along with the pointer. The size of the resulting RPC wire format is the same as that of InterWeave, because MIPs in InterWeave are strings, longer than four bytes. In addition, the RPC overhead for structures with doubles inside is high in part because `rpcgen` does not inline the marshaling routine for doubles.

The data management costs for the InterWeave server are much lower than that on the client in all cases other than `pointer` and `small_string` because the server maintains data in wire format. The high costs for `pointer` and `small_string` stem from the fact that strings and MIPs are of variable length, and are stored separately from their wire format blocks. However, for data structures with a reasonable number of pointers and small strings such as `mix`, the server cost is still comparable (see [15] for the results).

4.2. Modifications at Different Granularities

Figure 5 shows client and server diffing overhead as a function of the fraction of a segment that has changed. In all cases the segment in question consists of a 1MB array of integers. The X axis indicates the distance in words between consecutive modified words. Ratio 1 indicates that the entire block has been changed. Ratio 4 indicates that every 4th word has been changed, etc.

The "client collect diff" cost has been broken down into "client word diffing"—word-by-word comparison of a page and its twin—and "client translation"—converting the diff to wire format. (Values on these two curves add together to give the values on the "client collect diff" curve. We show them as line graphs for the sake of clarity.) There is a sharp knee for word diffing at ratio 1024. Before that

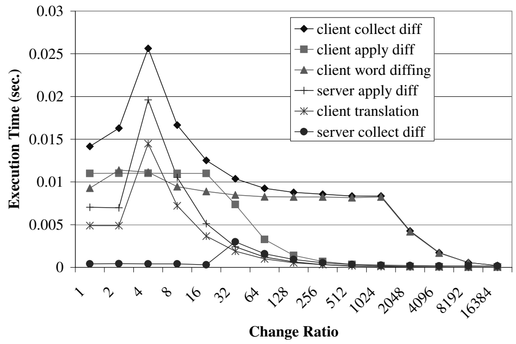

Figure 5. Diff management cost as a function of modification granularity (1MB total data).

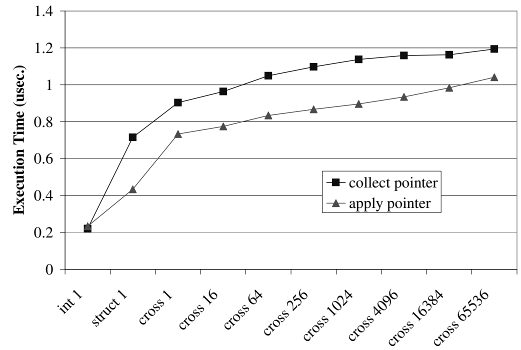

Figure 6. Pointer swizzling cost as a function of pointed-to object type.

point, every page in the segment has been modified; after that point, the number of modified pages decreases linearly. Due to the artifact of subblocks (16 primitive data units in our current implementation), the "server collect diff" and "client apply diff" costs are constant for ratios between 1 and 16, because in those cases the server loses track of fine-grain modifications and treats the entire block as changed. The jump in "client collect diff", "server apply diff", and "client translation" between ratios 2 and 4 is due to the loss of the *run splicing* optimization described in Section 3.3. At ratio 2 the entire block is treated as changed, while at ratio 4 the block is partitioned into many small isolated changes. The cost for "word diffing" increases between ratios 1 and 2 because the diffing is more efficient when there is only one continuous changed section.

Figures 4 and 5 show that InterWeave is efficient at translating both entirely changed blocks and scattered modifications. When only a fraction of a block has changed, InterWeave is able to reduce both translation cost and required bandwidth by transmitting only the diffs. With straightforward use of an RPC-style system, both translation cost and bandwidth remain constant regardless of the fraction of the data that has changed.

4.3. Pointer Swizzling

Figure 6 shows the cost of swizzling ("collect pointer") and unswizzling ("apply pointer") a single pointer variable. This cost varies with the nature of the pointed-to data. The "int 1" case represents an intra-segment pointer to the start of an integer block. "Struct 1" is an intra-segment pointer to the middle of a structure with 32 fields. The "cross #n" cases are cross-segment pointers to blocks in a segment with n total blocks. The modest rise in overhead with n reflects the cost of search in various metadata trees. Performance is best in the "int 1" case, which we expect to be representative of the most common sorts of pointers. However, even for moderately complex cross-segment pointers, InterWeave can swizzle about one million of them per second.

4.4. Translation Costs for a Datamining Application

In an attempt to demonstrate the benefits that an application can harvest from wire-format diffing, pointer-swizzling, and relaxed coherence models, we have measured communication costs in a locally developed datamining application. The application performs incremental sequence mining on a remote database of *transactions* (e.g., retail purchases). Details of the application are described elsewhere [5].

Our sample database is generated by tools from IBM research [12]. It includes 100,000 customers and 1000 different items, with an average of 1.25 transactions per customer and a total of 5000 item sequence patterns of average length 4. The total database size is 20MB.

In our experiments, we have a database server and a datamining client. Both are InterWeave clients. The database server reads from an active, growing database and builds a summary data structure (a lattice of item sequences) to be used by mining queries. Each node in the lattice represents a potentially meaningful sequence of transactions, and contains pointers to other sequences of which it is a prefix. This summary structure is shared between the database server and the mining client in an InterWeave segment. Approximately 1/3 of the space in the local-format version of the segment is consumed by pointers.

The summary structure is initially generated using half the database. The server then repeatedly updates the structure using an additional 1% of the database each time. As a result, the summary structure changes slowly over time. Given the statistical nature of the data, the datamining client need not always access the most recent copy of the summary structure to provide satisfactory results to mining queries: it can save translation and communication overhead by using InterWeave's relaxed coherence models.

Figure 7 shows the total bandwidth requirement as the client relaxes its coherence model. The leftmost bar rep-

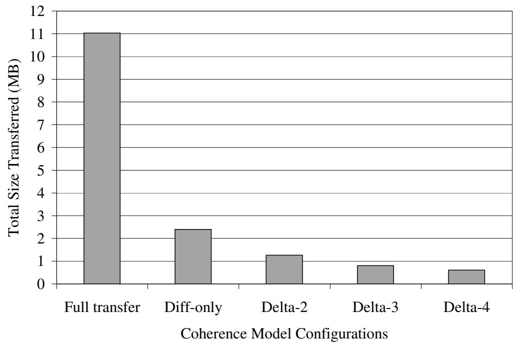

Figure 7. Total bandwidth requirement of the datamining application.

resents the bandwidth requirement if the client needs to transfer the whole summary structure each time a new version is available at the server. The second bar shows the bandwidth requirement with wire-format diffs. The right three bars show the bandwidth requirements as the client relaxes its coherence model to let its summary structure be updated every second, third, or fourth version (Section 3.2). We can see that using the wire-format diff to update the client's cache can reduce bandwidth requirements by a total of 80%. Additional reductions can be achieved if the coherence model is further relaxed.

4.5. Ease of Use

We have implemented several additional applications on top of InterWeave. One particularly interesting example is a stellar dynamics code called Astroflow, developed by colleagues in the department of Physics and Astronomy, and modified by our group to take advantage of InterWeave's ability to share data across heterogeneous platforms.

Astroflow is a computational fluid dynamics system used to study the birth and death of stars. The simulation engine is written in Fortran, and runs on a cluster of four AlphaServer 4100 5/600 nodes under the Cashmere [14] S-DSM system. As originally implemented, it dumps its results to a file, which is subsequently read by a visualization tool written in Java and running on a Pentium desktop. We used InterWeave to connect the simulator and visualization tool directly, to support on-line visualization and steering. The changes required to the two existing programs were small and isolated. We wrote an IDL specification to describe the shared data structures and replaced the original file operations with access to shared segments. No special care is required to support multiple visualization clients. Moreover, the visualization front end can control the frequency of updates from the simulator simply by specifying a temporal bound on relaxed coherence [5].

Performance experiments [5] indicate that InterWeave imposes negligible overhead on the existing simulator. We also believe the InterWeave version to be significantly simpler, easier to understand, and faster to write than a hypothetical version based on application-specific messaging. Our experience changing Astroflow from an off-line to an on-line client highlighted the value of middleware that hides the details of network communication, multiple clients, and the coherence of transmitted data.

5. Related Work

InterWeave finds context in an enormous body of related work—far too much to document thoroughly in this paper. We focus here on systems that address heterogeneity, leaving conventional S-DSM and distributed shared object systems out of the discussion.

Toronto's Mermaid system [17] allowed objects to be shared across more than one type of machine, but required that all data in the same VM page be of the same type, and that objects be of the same length on all machines, with the same byte offset for every subcomponent.

CMU's Agora system [2] supported sharing among more loosely-coupled processes, but in a significantly more restricted fashion than in InterWeave. Pointers and recursive types were not supported, and all shared data had to be accessed indirectly through a local mapping table.

Systems that support process or thread migration among heterogeneous computers such as Emerald [13] and Tui [11] employ data marshaling for mobility, but they do not employ data caching. As a result, they do not face the challenges of representing diffs in wire format. They also depend on compiler support to extract data type information.

Smart RPC [7] is an extension to conventional RPC that allows argument passing using call-by-reference rather than deep copy call-by-value. The biggest difference with respect to InterWeave is that Smart RPC does not have a shared global space with a well-defined cache coherence model. Smart RPC invalidates the cache after each RPC *session* (initial client request and nested callbacks), while InterWeave allows cache reuse.

ScaFDOCS [8] is an object caching framework built on top of CORBA. As in Java RMI, shared objects are derived from a base class and their *writeToString* and *readFromString* methods are used to serialize and deserialize internal state. CASCADE [6] is a distributed caching service, structured as a CORBA object. Both ScaFDOCS and CASCADE encounter a fundamental limitation of CORBA's reference model: because everything is an object with no exported data members, every use of a reference parameter incurs a callback. They also suffer from the lack of diffs: small changes to large objects still require large messages.

Object Oriented Databases (OODBs) such as Thor [9] allow objects to be cached at client front ends, but they usually neither address heterogeneity nor attempt to support a shared memory programming model.

InterAct [10] is an object-based system that uses relaxed coherence to support distributed sharing, but requires shared data to be accessed through C++ templates. InterWeave provides a more transparent interface, allowing ordinary reads and writes to shared data once mapped.

6. Conclusions and Future Work

We have described the design and implementation of a middleware system, InterWeave, that allows processes to access shared data transparently and efficiently across heterogeneous machine types and languages using ordinary reads and writes. The twin goals of convenience and efficiency are achieved through the use of a wire format, and accompanying algorithms and metadata, rich enough to capture machine- and language-independent diffs of complex data structures, including pointers or recursive data types. InterWeave is compatible with existing RPC and RMI systems, for which it provides a global name space in which data structures can be passed by reference. Experimental evaluation demonstrates that automatically cached, coherent shared state can be maintained at reasonable cost, and that it provides significant performance advantages over straightforward (cacheless) use of RPC alone.

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. We are incorporating transaction support into InterWeave and studying the interplay of transactions, RPC, and global shared state.

References

[1] 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*, pages 261–271, San Antonio, TX, Feb. 1997.

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

[3] 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.

[4] D. Chen, C. Tang, S. Dwarkadas, and M. L. Scott. JVM for a Heterogeneous Shared Memory System. In *Proc. of the Workshop on Caching, Coherence, and Consistency (WC3 '02)*, New York, NY, June 2002. Held in conjunction with the *16th ACM Intl. Conf. on Supercomputing*.

[5] 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.

[6] 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.

[7] K. Kono, K. Kato, and T. Masuda. Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers. In *Proc. of the 14th Intl. Conf. on Distributed Computing Systems*, pages 142–151, Poznan, Poland, June 1994.

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

[9] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing Persistent Objects in Distributed Systems. In *Proc. of the 13th European Conf. on Object-Oriented Programming*, pages 230–257, Lisbon, Portugal, June 1999.

[10] S. Parthasarathy and S. Dwarkadas. Shared State for Distributed Interactive Data Mining Applications. *Intl. Journal of Distributed and Parallel Databases*, 11(2):129–155, Mar. 2002.

[11] P. Smith and N. C. Hutchinson. Heterogeneous Process Migration: The Tui System. *Software — Practice and Experience*, 28(6):611–639, 1998.

[12] 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.

[13] B. Steensgaard and E. Jul. Object and Native Code Thread Mobility among Heterogeneous Computers. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 68–77, Copper Mountain, CO, Dec. 1995.

[14] 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*, pages 170–183, St. Malo, France, Oct. 1997.

[15] 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.

[16] P. R. Wilson. Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware. In *Proc. of the Intl. Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, France, Sept. 1992.

[17] 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.