Smart Remote Procedure Calls: Transparent Treatment of Remote Pointers

Kenji KONO† Kazuhiko KATO‡ Takashi MASUDA†

† Department of Information Science
Faculty of Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
Email: {kono,masuda}@is.s.u-tokyo.ac.jp

‡ Institute of Information Sciences
and Electronics
University of Tsukuba
Tsukuba, Ibaraki 305, Japan
Email: kato@is.tsukuba.ac.jp

Abstract

Remote procedure call (RPC) systems have been proven to be a practical basis for building distributed applications. The RPC technique abstracts a typical communication pattern to an ordinary procedure call. Compared with an ordinary procedure call, however, the conventional RPC technique has one evident restriction; pointers (addresses) cannot be passed to remote procedures without the explicit and nontrivial programming effort. This paper presents a method that eliminates this restriction. The method enables transparent treatment of pointers in RPC by combining three key techniques: virtual memory manipulation, pointer swizzling, and coherency protocol. The experiments performed using an implementation of the method show that the method provides performance that is scalable to the access ratio of the remotely referenced data.

1 Introduction

Remote procedure calls (RPCs) are a key technique for building distributed systems and applications. They provide message-passing semantics for the procedure calls found in most programming languages. Most RPC systems provide a generator of code that performs most of the communication-specific operations at runtime. Those programs, called stubs, make communication largely transparent to the programmer; remote procedures can be written and used in almost the same way as local procedures. Hardware heterogeneity can also be handled by this approach. The arguments of a remote procedure are encoded into a canonical data representation at marshaling time, and the canonical representation is decoded at unmarshaling time to obtain the internal representation of the other machine. Including these encoding and decoding operations in the stubs of both sites makes it possible to preserve the data types in a heterogeneous environment.

The practicality of this approach is well recognized, and used as the basis for many experimental [3, 2, 9, 8] and commercial [6, 7, 17, 18] distributed systems. However, conventional RPC systems have a crucial restriction: only certain data types can be used as the arguments of a remote procedure. For example, pointers or higher-order functions cannot be used directly as arguments. In ordinary programming languages, it is common for programmers to pass pointers (addresses) to subroutines as arguments, but in the conventional RPC systems, it is either not permitted at all or must be controlled by the programmer. To circumvent this limitation, the programmer must, for example, write callback routines to dereference remote pointers or to call remote functions passed as arguments.

This paper describes a method for transparent support of remote pointers in RPC. It allows remote pointers to be used in the same way as local pointers without assuming any linguistic support. The method gives programmers the illusion that pointers can refer freely to data in other address spaces. The illusion is not restricted to the source code level; once a remote data is referenced, it is cached in the local address space and the runtime cost to access it is exactly the same as the cost to access ordinary local data, except for the cost to write back the modified cached data. The key issues to realize the transparent treatment of remote pointers are threefold:

- efficient detection of dereferencing of remote data,
- transparent treatment of remote pointers, and
- protocol that guarantees the coherency of address spaces.

To deal with these, we incorporated three techniques into the RPC: virtual memory manipulation, pointer swizzling, and coherency protocol.

Virtual memory manipulation technique takes advantage of the memory management hardware widely available in current computing environments. Modern operating system kernels such as Mach and SunOS provide primitives for user-level program control of page access to virtual memory and page-fault handling. This enables efficient and transparent detection of the first request to access remote data without modification of the operating system kernels.

Pointer swizzling is an address translation technique that has been used for persistent objects and persistent programming languages [4, 13, 10]. Our method applies pointer swizzling to allow programmers to describe the manipulation of remote pointers in the same way as local pointers. The combination of pointer swizzling and virtual memory manipulation was first proposed by Paul Wilson [16] as an efficient method of implementing larger address spaces than the word size of the available hardware in a non-distributed environment. The method described in this paper uses his method extensively to deal with distributed and heterogeneous address spaces.

To make the semantics of RPCs as close as that of the ordinary procedure calls, the method includes a runtime protocol that guarantees coherency among the address spaces involved in an RPC session. When the transferred remote data is modified in an address space (including memory allocation and release operations), the runtime system properly reflects the modifications in the original data, which may be in another address space.

The rest of this paper is organized as follows. Section 2 addresses the problems involved in implementing transparent treatment of pointers in RPC. Section 3 describes our method. Section 4 presents experimental results on the proposed method. Section 5 compares the method with related work. Section 6 concludes the paper.

2 Problems

Before presenting our approach, let us examine the problems posed by handling pointers with the conventional RPC techniques.

2.1 Eager and Lazy Methods

One straightforward way to pass a pointer to a remote procedure is to take the closure of the pointer on the caller side and pass it to the remote procedure as an input RPC argument. This method is *eager* in the sense that the data is transferred prior to the request to access it in the remote procedure body. The eager method is easily dealt with by the stub generator of RPC systems since stub generation for the method is not complicated. Most dynamic data structures that use pointers have recursive structures in their type definitions, so the generator simply generates the programs that recursively call one another. Indeed, Sun Microsystems' rpcgen system [15] passes recursive data structures such as lists or trees in this way. This is useful if the data pointed to is relatively small. However, it is often large. Consider the case that a large body of data is organized as an array or a binary tree, and access to only some portion of the data is required by the remote procedure. Marshaling the whole tree and sending it to the remote procedure would terribly increase the execution overhead. To avoid this, an experienced programmer might abandon the use of remote pointers and develop a caller-callee protocol to pass only the required portion of the tree.

Another approach to passing pointers as arguments in RPCs is to use the *callback* mechanism supported by many RPC systems. Callback means that a callee remotely calls its caller. Whenever a remote pointer must be dereferenced during the execution of a callee program, the callee calls back the caller with a request to pass the contents of the pointer. In this way, the pointer contents are passed by the on-demand or *lazy* method. This approach is suitable when a relatively small portion of a large amount of data is accessed in a remote procedure. When a relatively large portion of the data is accessed, the approach is less attractive, since the increased number of callbacks reduces execution performance. Also, a naive implementation of this approach might perform callbacks whenever a pointer is dereferenced, even if the pointer has already been dereferenced.

A possible solution for the above-mentioned problems is to "cache" a remote data. The referenced data is transferred from the caller to the callee when the data is first requested. The callee reuses the transferred data each time it must be accessed. The caching effect will probably reduce execution time because the callee accesses the locally cached data and remote data access is minimized. This approach is attractive but its implementation poses significant problems. How can the first access to remote data be distinguished from subsequent accesses with low overhead? Distribution transparency on the source code level can be accomplished in a higher degree? When the callee re-

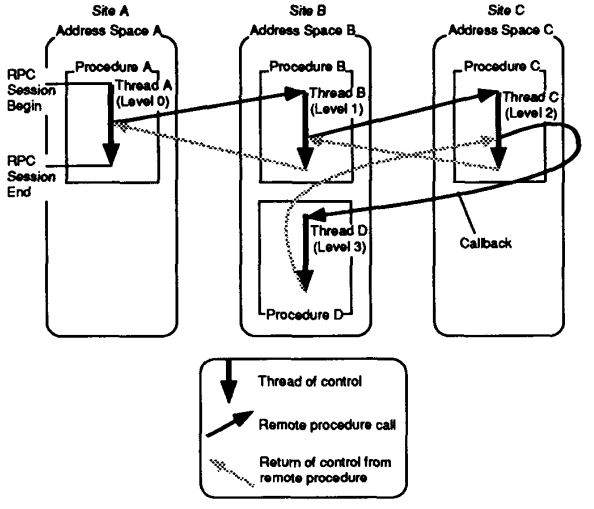

Figure 1: RPC Execution Model.

quires modification of the cached data, how can the coherency between the cache and the original data be maintained? The next section describes our approach to these problems.

3 Method

After considering the problems mentioned in the previous section, we developed a caching method that combines the merits of the lazy and the eager methods. Section 3.1 defines a fundamental RPC model and terminology. Section 3.2 describes the basic part of the proposed method. Section 3.4 describes the incorporation of eagerness and nested RPCs. Section 3.4 presents the coherency protocol. Section 3.5 presents a way to handle remote memory allocation and release.

3.1 Fundamental RPC Model and Terminology

We begin by describing the RPC execution model for which our method was developed (Fig. 1). A program is executed by a thread in an address space. When a thread calls a remote procedure, the following sequence is executed.

1. The execution of the thread is blocked until the output arguments are returned from the remote procedure.

2. A thread is initiated on the callee site to execute the called remote procedure.

3. The initiated thread on the callee terminates when it returns the output arguments to the caller.

A *ground* thread is one whose execution is not initiated by an RPC (perhaps a user has initiated it). In Fig. 1, a ground thread executes procedure A. A ground thread must declare the beginning and the end of an *RPC session*. The concept of an RPC session is needed to determine the period for which the runtime system guarantees to respond to remote data references and to maintain the coherency of the cached data. For example, assume that remote procedure B returns to procedure A a pointer that references data in address space B. The remote pointer is effective only within the session; after the RPC session, the remote pointer has no meaning.

RPCs can be nested. In Fig. 1, three RPCs are issued in one RPC session in a nested fashion. *Callback* is also allowed, which means a callee remotely calls its caller. We maintain the usual synchronous property of RPC—that is, only a single thread is active in an RPC session, even when several sites participate in the session.

3.2 Basic Method

Our method is a fairly sophisticated one; it incorporates the eager, lazy and caching techniques into an RPC system. For ease of understanding, we first describe the lazy portion of the method with caching, and then describe the eager portion. To concentrate our discussion on transparent treatment of remote pointers, we assume that the reader is familiar with conventional RPC stub-generation techniques (See [3] and [14] for explanation of this technique). To simplify the explanation, we describe the case where a pointer that references data on the caller side is passed from the caller to the callee. Then we mention the general case where pointers are passed freely.

We begin by introducing some notions. Generally, in a single address space, a pointer can designate a location valid only in the address space. To allow the passing of pointers beyond the boundaries of an address space, pointer definition must be extended to the entire distributed system. Thus, we introduce the concept of a *long-format pointer* (long pointer for short). We then term the non-long format pointer an *ordinary pointer*. A long pointer is composed of three elements:

- an address space identifier defined in the distributed environment (typically a pair consisting of a site ID and a process ID in the site),
- an address valid within the address space, and

- a data type specifier that specifies the type of the data referenced by this pointer.

To locate a data in an distributed environment, we require an address space identifier and an address valid within the address space. Additionally, a data type specifier is necessary for the system to be heterogeneous. Data type is essential to interchange of internal data representations among different architectures.

We assume that the system can obtain an actual data structure from a data type specifier by querying a database that serves as a network name server. All data referenced by long pointers are assumed to be located in the heap area under the system control.

Since generally-available hardware can only deal with ordinary pointers, long pointers must be translated into ordinary pointers, at least until the hardware uses them. We call the translation from a long pointer into the corresponding ordinary pointer *pointer swizzling*, and we call the reverse translation *pointer unswizzling*.

When a remote pointer is passed as an argument of a remote procedure, the pointer is unswizzled on the caller side. This translation is coded in the caller stub for the remote procedure. On the other hand, the callee stub for the remote procedure includes the code that swizzles the pointer. Here, a question arises. What address should the swizzled pointer in the callee reference? Ordinary pointers in the callee address space can only reference addresses within that address space, but referenced data exists only on the caller side at this time. The solution is that when the callee receives a long pointer from the caller, the callee allocates for the referenced data a *protected* page area. Protected page area means that the area is protected from access (including read or write) by the page-protection mechanism of the memory management unit (MMU) hardware. The allocation determines the location to which the referenced data *will* be copied if the protected page area must be accessed. The determined location is used as the address into which the long pointer is swizzled. Note that the page contains no data at this time. The data to fill the page are transferred when the callee tries to access the page as described bellow.

The above situation is illustrated in Fig. 2, where two pointers, A and B, are transferred from the caller to the callee. One protected page can include several sets of referenced remote data. In the figure, two sets of data are allocated to the protected page. The data allocated to a protected page area is transferred later when necessary.

MMU detects the first access to data allocated to a

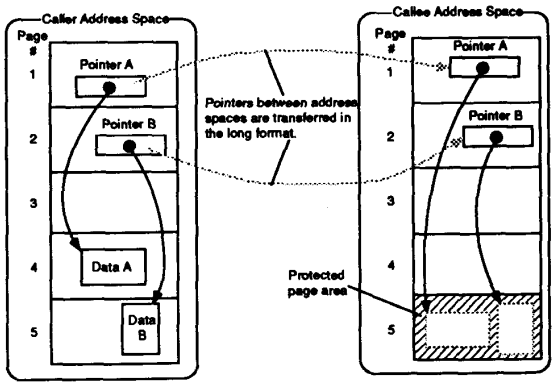

Figure 2: Just after pointers A and B are swizzled in the callee address space.

protected page, and raises an access-violation exception. The operating system kernel is informed a priori that the runtime system handles the exception. Catching the exception, the handler determines at which location the exception was raised (modern operating systems such as Mach or SunOS provide these functions). The requested data is transferred from the caller's address space to the callee's at this time. All of the other data allocated to the page must be transferred at this time, because once the access protection of the page is released, the first access to the other data in the page can no longer be detected.

The runtime system maintains a *data allocation table* that records what data should be transferred from remote address spaces. The entries of the table are the page number, the offset within the page, and a long pointer. For the example shown in Fig. 2, the data allocation table would be like Table 1. The runtime system refers to the data allocation table and then communicates with the other runtime systems that manage the original data to request the sending of the data.

Table 1: Data allocation table.

page #	offset within the page	long pointer
5	offset_1	A
5	offset_2	B

Figure 3 illustrates the data transfers for this example. After the data transfer, the runtime system directs the operating system kernel to release the access protection of the page. The runtime system then resumes the thread that caused the access-violation exception. Since the transferred data is cached in the

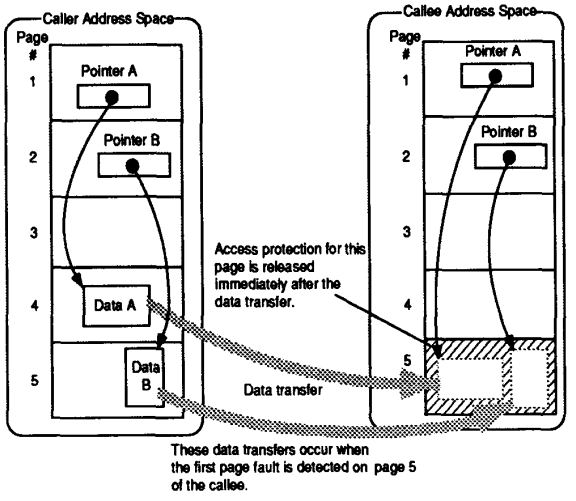

Figure 3: When a protected page is accessed, all the data allocated to the page are transferred with their representations properly translated.

page, the subsequent accesses to the data are the same as accesses to local data.

On the data transfers, data representations must be encoded and decoded to preserve their data types in a heterogeneous environment. We can use the standard methods except for the case of pointers, which must be unswizzled and swizzled as described above when the transferred data structures include them.

3.3 Eagerness and Nested RPCs

Until now we have concentrated our description on the lazy portion of the method. Next we describe how eagerness is incorporated. As mentioned in Section 2, eagerness concerns the timing of the transfer of data referenced by remote pointers. Generally a remote pointer can be considered as a capability to access the data referenced by it; the data might, or might not, be accessed in the address space that received the pointer. We can expect eager data transfer to provide better execution performance than lazy transfer, when most of the remotely referenced data is accessed, since fewer communications are required.

We introduce eagerness to the method by transferring a certain depth of the *transitive closure* of a pointer when the pointer is transferred to a remote address space. On the transfer, of course, proper encoding, decoding, unswizzling, and swizzling must be done according to the data types in a way similar to the lazy case described above. There are several alternative algorithms and parameters for taking a certain depth of transitive closure. Our current implementation uses the breadth-first traverse algorithm with the maximum amount of the traversed data explicitly specified by the user. We discuss this issue more in Section 6.

Above, we have described mainly the case where pointers are passed between two distinct address spaces. The explained method works well even when RPCs are *nested*—that is, a program calls a remote procedure P_1 and in its execution P_1 calls another remote procedure P_2, and so on. That is made possible by the long pointers and the data allocation table; whenever required, an ordinary pointer can be translated into and from a long pointer by reference to the data allocation table. Therefore, pointers can be freely passed in the same way as described above, even when RPCs are nested, by the combination of unswizzling and swizzling.

3.4 Coherency Protocol

In general, all the caching techniques that permit update operations inherently have the problem of coherency between original data and their copies. In our method, the system must maintain the coherency of the cached data in all address spaces that participate in an RPC session.

Our coherency protocol takes advantage of a virtue of RPC, synchronous communication. In an RPC session, even when the session includes nested RPCs, there is only one active thread of control.[1] Therefore, coherency must be guaranteed only for the execution of the active thread.

Consider the case where thread C in Fig. 1 requires access to data which is originally located in address space A, and cached and modified in address space B. If the coherency protocol we are going to describe is not used, thread C will refer to the data in address space A and the modification performed in address space B will be not effective. To prevent this, our coherency protocol forces the runtime system to send all the "dirty" data in the cache when the activity of threads moves beyond address spaces—that is, when input arguments are passed to a remote procedure and output arguments are passed back to the caller. When a thread issues an RPC and the address space of the thread has dirty caches, the protocol transfers all

[1] Some people might say that this is a limitation of RPC, but this work concentrates on the issue of eliminating the restriction on the treatment of pointers.

the cached data allocated to dirty cache pages to the callee. The reason why the protocol works well with the method described hitherto is that the protocol can be seen as a case of the eager data transfer described in Section 3.3; the protocol tells the runtime system to send eagerly the data in the dirty cache pages when the input arguments of the RPC are sent.

To efficiently detect data modification, we can again make use of the MMU hardware. When remote data are copied in the cache area of an address space, the runtime system sets read-only access permission to the pages by using MMU. When a thread tries to modify one of the write-protected pages, MMU raises an access-violation exception, and lets the runtime system know which page is being modified.

When a nested RPC occurs, the protocol forces all the data in all the dirty pages to be passed between address spaces. (Note that dirtiness can be detected by page-grain.) Those dirty pages, which we call *modified data set*, must be written back to the original location. Otherwise, it would continue to increase monotonically, decreasing the runtime performance. When should the modified data set be written-back? We assume that there would be a "working set" in distributed computation as well as centralized computation, and use the concept of an RPC session to determine that working set. Each site involved in the RPC session keeps all the cached data until the ground thread declares the end of the session. Up to the end of the session, the modified data set is passed among the address spaces with the transition of thread activation. Thus, each address space in the session can always see the correct working set with minimum page transferring cost. At the end of the session, the runtime system managing the address space of the ground thread performs the following two tasks:

- Examine the modified data set, and write back each modified page to the original address space.
- Multicast a message to the address spaces concerning the RPC session to invalidate all the cached data.

3.5 Remote Memory Allocation and Release

The system supports transparent remote memory allocation and release operations, and provides the following two primitives (we use the notation of ANSI C; `void*` designates a data type of pointers):

- `void* extended_malloc(address_space_ID, data_type_ID)`

  This primitive allocates a memory area in the address space specified by `address_space_ID` to store data whose type is specified by `data_type_ID`. It returns a swizzled pointer valid in the address space in which it is issued.

- `void extended_free(void* p)`

  This primitive releases the memory area allocated for the data referenced by pointer `p`. Note that `p` may reference data whose original location is not in the address space in which it is issued.

When a thread issues these primitives, the runtime system of the address space of the thread performs the required memory allocation or release operation on the cached memory area in the address space. The operation must be reflected in the data allocation table in the address space. Also, the data area must be allocated or released in the original address space specified by `address_space_ID`. One straightforward timing for allocation and release of the data in the original space is upon each issuing of the allocate and release primitives. However, this would degrade the runtime performance terribly, considering that remote allocation and release of hundreds of data sets may be requested consecutively. Our solution to this problem is that the runtime system batches the memory allocation and release operation requests to the original address spaces. The batch operations are performed when the activity of the thread moves to another address space. This restrains the runtime overhead, since the number of communications is reduced; a single message to an address space can include multiple requests of data allocations and releases to the address space.

4 Experiments

To validate the usefulness of the proposed method, we implemented an RPC system based on it. The system was created on SunOS 4.1.1 running on Sun SPARC (28.5 MIPS) workstations. The stations have 48 Mbytes of main memory and are connected by a 10 Mbps Ethernet network. The system uses the TCP/IP network protocol and specifies the `TCP_NODELAY` option in the socket system call so that small packets are sent as soon as possible. As a canonical data representation, the system uses XDR (eXternal Data Representation) [12], which guarantees the transformation of basic data types such as integers, floating point numbers, and strings between CPUs with different architectures. We used the XDR library provided

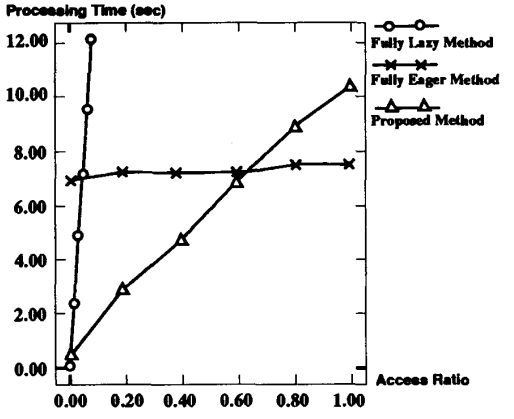

Figure 4: Comparison of the Three Methods. X-axis: (number of nodes accessed in the callee)/(total number of nodes); Y-axis: processing time (seconds).

with the SunOS. Although the experimented system consists of only a few SPARC stations, the system was carefully implemented so as to deal with heterogeneity. Therefore, the experimental results reflect the heterogeneity overhead, such as data representation conversion.

4.1 Comparison between the Methods

Here, we compare the performance of our method and the eager and lazy methods described in Section 2, which we call respectively *fully eager* and *fully lazy* methods.

The experimental subject was a search of a complete binary tree. Each node of the tree has 16 bytes (two 4-byte pointers and 8-byte data). Initially, a complete binary tree of 32,767 nodes was created in the caller address space. We measured the average time required to process one remote procedure call that retrieves the tree, varying the number of nodes retrieved in the *callee*. The nodes of the tree were visited in a *depth-first* manner until the ratios of the number of visited nodes to the total number of the nodes reached the ratio indicated by X-axis in Fig. 4. The tree was not sent back to the caller since no modification was performed. The RPCs were performed by the following three methods.

- With the fully eager method, the caller sent the whole tree (524,272 bytes) to the callee, which accessed the tree locally.
- With the fully lazy method, the caller sent to the callee a pointer to the root of the tree, and the callee retrieved the tree by performing callbacks for each dereferencing of the pointers in the tree.
- With the proposed method, the callee received a pointer to the root of the tree and retrieved the tree as if it were in the callee's address space. When a dereferencing request to a pointer caused a page fault in the callee, the transitive closure of the pointer was retrieved in a breadth-first manner in the caller. We set the closure size to 8192 bytes in this experiment.

Figure 4 shows the experimental results. With the fully eager method, the processing time is nearly constant, because the whole tree was sent once just before the execution of the remote procedure body.

With the fully lazy method, the processing time is obviously bad. This is because of the increased number of callbacks. Figure 5 is the same as Fig. 4 except that the Y-axis designates the number of callbacks. By comparing Fig. 4 and Fig. 5, we can see that the number of callbacks, which take much time, dominates the processing time of the fully lazy method. In this method, the data transfer granularity was too fine to fully utilize the network bandwidth. The fully lazy method is expected to show good performance when a small portion of the large data is accessed (for example, retrieval of a hash table).

The proposed method provides the best processing time of the three methods for access ratios between 0.0 and 0.6. This owes to the combined effects of the lazy, eager and caching techniques. With the proposed method, only limited portions of the tree were sent to the callee, as is clearly observed in Fig. 5. The improved performance relative to the fully eager method was obtained because relatively small portions of the remote data (tree) were transferred. When relatively large portions of the data were accessed ((access ratio) > 0.6), the increase in the number of communications rendered the proposed method disadvantageous.

4.2 Closure Size

In the proposed method, the closure size parameter, which specifies the degree of transitive closures, plays an important role. When the parameter is set to zero, the behavior is similar to the fully lazy method. When set to infinity, the behavior resembles the fully eager method. We did an experiment using almost the same experimental subject to determine how the parameter affected execution performance. In the experiment a complete binary tree created on a caller was remotely searched in a callee. We measured the average time to process one remote procedure call in which the nodes of the tree were *remotely* visited from the root to the leaves for 10 times. The reason for repeating searches

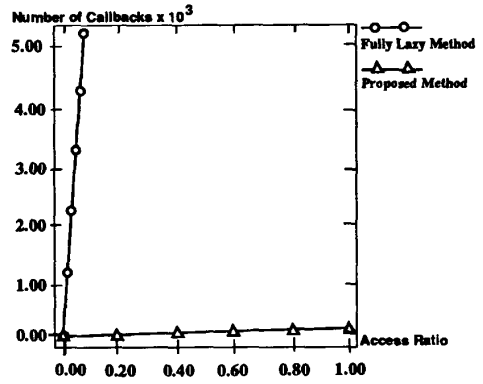

Figure 5: Comparison between the Lazy Method and the Proposed Method. X-axis: (number of nodes accessed in the callee)/(total number of nodes); Y-axis: number of callbacks.

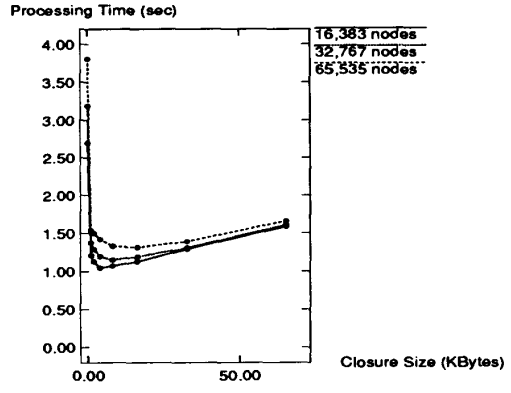

Figure 6: Relationship between the Closure Size and Processing Time.

is to increase the effect of caching; nodes in the upper level will be reused in the subsequent searches. Figure 6 shows the results. The performance is not good when the closure size is too small. In the experiment the optimal closure sizes are relatively small: respectively 4, 8 and 16 Kbytes for 16383, 32767 and 65535 nodes. This is because of the nature of the experimental subject. As the number of nodes in the tree increases exponentially, the larger closure could not effectively carry the retrieved data.

4.3 Update

Finally we examined how update operations on the data referenced by remote pointers affected the system performance. The experimental subject used was the same as that in the subsection 4.1; a complete binary tree residing on a caller was *remotely* accessed

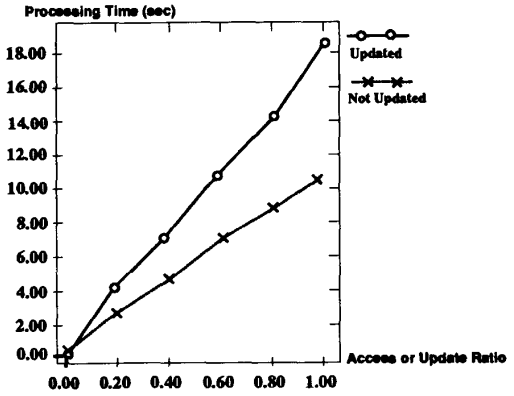

Figure 7: Update Performance. X-axis: (number of nodes updated in the callee)/(total number of nodes) in the updated case and (number of nodes updated in the callee)/(total number of nodes) in the not updated case; Y-axis: processing time (seconds).

in a callee. We set the closure size parameter to 8192 bytes. In Fig. 7, the solid line represents the case where the data in the tree were updated in the ratio indicated by the X-axis, while the dotted line represents the case where the data were not updated, but visited in the ratio indicated by the X-axis. The access patterns in both cases were the same except for updates so that we could measure overheads incurred by updates. The result is favorable for the following two reasons. First, the increase in the processing time for the updated case is scalable to the update ratio. Second, each processing time for the update case is just twice of that of the not-updated case. This is reasonable, since the update in the remote procedure body requires at least two page accesses: one for reading and the other for writing-back the updated data.

5 Related Work

This section describes related work that deals with pointers in distributed systems. We mentioned in Section 2 the conventional approaches to dealing with remote pointers in RPC systems.

5.1 Linguistic Approach

Some programming languages designed for distributed systems have the functions for dealing with remote pointers. The approaches to the treatment of remote pointers in distributed programming languages are classified into those that provide a data type that can logically represent "links" between other data, and

those that provide a remote address reference capability. A typical example of the former is the Orca distributed programming language [1]. In that, the links are treated in a very different way from the ordinary pointers that can be directly interpreted by the CPU. Transparency to the programmers is abandoned, in favor of simpler implementation with respect to remote pointers. Typical examples of the latter approach are CLAM [5] and Distributed C [11]. The developers of those languages incorporated RPC and remote pointer functions into existing programming languages, C++ and C, respectively. Those systems attain a fairly high degree of distribution transparency on the source code level. The technique proposed in this paper is superior in the following two ways. First, it provides transparent treatment of remote pointers independently of individual programming languages; programmers need not be aware that a pointer is local or remote. An application programmer of CLAM or Distributed C has to provide additional description when using a remote pointer. Second, the proposed method automatically caches the remote data. This should lead to better execution performance than either of the two languages.

5.2 Heterogeneous Distributed Shared Memory

The proposed method and distributed shared memory (DSM) systems have some common properties. First, both techniques enable access to remote data as if they were local data. Second, both techniques use a virtual memory manipulation technique as a central part. By nature, DSMs were designed for homogeneous hardware environments, but some recent DSM systems such as Mermaid [19] were designed to deal with hardware heterogeneity. Hence we can say that, third, both the proposed system and some DSM systems can share data between distributed address spaces of heterogeneous architectures.

It might be said that the proposed method is subsumed by the (heterogeneous) DSM systems, since the proposed method was designed for RPCs—a typical synchronous communication pattern. The proposed method, however, has the following several remarkable features lacked by the DSM systems.

The proposed method takes advantage of the synchronous nature of RPCs in the design of the coherency protocol. The protocol of the proposed method does not have concurrency control, since at any time there is only one active thread in an RPC session. This makes the coherency protocol of the proposed method lighter than that of the DSM systems. Thus, if an application is satisfied by RPC communications, it can enjoy faster execution by using the proposed method.

Second, heterogeneity dealt with in the heterogeneous DSM systems is restricted. For example, Mermaid, one of most recently designed heterogeneous DSM systems, has the following limitations.

- All processors must use the largest main-memory word alignment. For instance, if one machine has a 64-bit CPU and the rest have 32-bit CPUs, all the data on the DSM must be aligned with 64-bits words.
- Every compiler must use the same data format. For instance, the memory usage map for a record type must be the same in every machine.

These limitations originate from the fact that the heterogeneous DSM systems physically share distributed memory. In contrast, the proposed method does not share distributed memory; it shares only the logical type of the shared data. Therefore, the proposed method does not have the above limitations. The limitations of the heterogeneous DSM systems would become severe for the creation of a practical heterogeneous distributed system, since all the architectures of the distributed system must be known in advance.

6 Conclusion

We have described a method for treating pointers in RPCs in a transparent way. The method combines three key techniques: virtual memory manipulation, pointer swizzling, and coherency protocol. Virtual memory manipulation is used to detect requests to access the data remotely referenced by pointers. Additionally, it contributes to automatic caching of the transferred referenced data. The pointer swizzling technique gives the programmers the illusion that the address space boundary does not exist in a distributed system; the programmers see that an ordinary pointer can be passed to any address space, and the dereferencing of the pointer works precisely the same way as when pointers are passed between procedures within a single address space. So as not to limit the data referenced by a remote pointer to read-only, we developed a coherency protocol for the RPC session. The protocol can effectively find modified data using virtual memory manipulation. The synchronous communication nature of RPC contributes to free the protocol from complicated and heavy concurrency controls.

There still remains a limitation concerning treatment of remote pointers in the proposed method; the

method does not support a remote pointer to a function. This limitation might not be negligible, since passing a pointer that references a function to remote procedure is one of the strongest motivations for using remote pointers and performing callbacks in the conventional RPC programming style. Ohori and Kato [14] recently developed a systematic stub generation method that provides for the programmers the illusion that any polymorphic higher-order functions can be passed among heterogeneous address spaces. Fortunately, their method and the method proposed in this paper do not conflict. Indeed, the authors are currently working on developing an RPC system that combines the two RPC techniques.

Some interesting research issues still remain. One is to develop a data allocation method for the cache area. As discussed in Section 3.2, when a pointer is transferred to a remote address space, the runtime system of the address space must determine where the data referenced by the pointer should be allocated to swizzle the pointer. The allocation method affects execution performance, since all the data in the (protected) page must be transferred at the time when access to the page is required. The worst situation is that all the data in the page are located at different computing sites. The current implementation uses a heuristic allocation strategy, with which all the data in a page is located in a single address space. This works well in many cases, but it increases the size of the working set when small amounts of data are transferred from an address space. Therefore we should develop a general allocation method to find the optimal tradeoff between working set size and number of communications.

Another issue is to develop an algorithm for optimizing the "shape" of the subset of the transitive closure of a pointer when the data referenced by the pointer is transferred eagerly. Precise estimation of the shape would minimize the number of communications, but it requires predetermination of the access request performed in the remote procedure body. One promising solution is to use suggestions provided by the programmer.

Acknowledgement

The authors would like to thank Atsushi Ohori for valuable discussions on various implementation issues on RPC.

References

[1] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek. Orca: A language for distributed object-based programming. *SIGPLAN Notices*, 25(5):17–24, may 1990.

[2] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz. A remote procedure call facility of interconnecting heterogeneous computer systems. *IEEE Trans. Software Engineering*, SE-13(8):880–894, Aug. 1987.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, Feb. 1984.

[4] R. G. G. Cattell. *Object Data Management—Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[5] D. L. Cohrs, B. P. Miller, and L. A. Call. Distributed upcalls: a mechanism for layering asynchronous abstractions. In *Proc. IEEE Int. Conf. on Distributed Computing Systems*, pages 55–62, 1988.

[6] J. R. Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls.* Sun Technical Reference Library. Springer-Verlag, 1990.

[7] R. Draves, M. Jones, and M. Thompson. MIG—the Mach interface generator. Technical report, Carnegie-Mellon University, Feb. 1988.

[8] P. B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Trans. Software Engineering*, SE-13(1):77–87, Jan. 1987.

[9] R. Hayes and R. D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Trans. Software Engineering*, SE-13(12):1254–1264, Dec. 1987.

[10] K. Kato, A. Narita, S. Inohara, and T. Masuda. Distributed shared repository: A unified approach to distribution and persistency. In *Proc. IEEE 13th Int. Conf. on Distributed Computing Systems*, pages 20–29, May 1993.

[11] K. Kato, A. Ohori, T. Murakami, and T. Masuda. Distributed C language based on a higher-order remote procedure call technique. In *Advances in Software Science and Technology*, volume 5, pages pp. 119–143. Academic Press, 1993.

[12] Sun Microsystems. *XDR: External Data Representation Standard.* RFC 1014, 20 pages, Jun. 1987.

[13] J. E. B. Moss. Working with persistent objects: to swizzle or not to swizzle. *IEEE Trans. Software Engineering*, 18(8):657–673, Aug. 1992.

[14] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Proc. 20th ACM Symp. on Principles of Programming Languages*, pages 99–112, Jan. 1993.

[15] SUN Microsystems. *SUN OS Reference Manual*, 1988.

[16] P. Wilson. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *ACM Computer Architecture News*, pages 6–13, Jun. 1991.

[17] XEROX Corporation. *Courier: the Remote Procedure Call Protocol*, Dec. 1981. XEROX system integration standard XSIS-038112.

[18] L. Zahn, T. H. Dineen, P. J. Leach, E. A. Martin, N. W. Mishkin, J. N. Pato, and G. L. Wyant. *Network Computing Architecture.* Prentice Hall, 1990.

[19] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Trans. Software Engineering*, 3(5):540–554, Sep. 1992.