Fast Heterogeneous Binary Data Interchange

Greg Eisenhauer and Lynn K. Daley
College of Computing
Georgia Institute of Technology

Abstract

As distributed applications have become more widely used, they more often need to leverage the computing power of a heterogeneous network of computer architectures. Modern communications libraries provide mechanisms that hide at least some of the complexities of binary data interchange among heterogeneous machines. However, these mechanisms may be cumbersome, requiring that communicating applications agree a priori *on precise message contents, or they may be inefficient, using both "up" and "down" translations for binary data. Finally, the semantics of many packages, particularly those which require applications to manually "pack" and "unpack" messages, result in multiple copies of message data, thereby reducing communication performance. This paper describes PBIO, a novel messaging middleware which offers applications significantly more flexibility in message exchange while providing an efficient implementation that offers high performance.*

1 Introduction

As distributed applications have become more widely used, they often need to leverage the computing power of a heterogeneous network of computer architectures. Modern communications libraries provide mechanisms that hide at least some of the complexities of binary data interchange among heterogeneous machines. The features and semantics of these packages are typically a compromise between what might be useful to the applications and what can be implemented efficiently.

For example, many packages, such as PVM[8] and Nexus[7], support message exchanges in which the communicating applications "pack" and "unpack" messages, building and decoding them field by field, datatype by datatype. Other packages, such as MPI[6], allow the creation of user-defined datatypes for messages and message fields and provide some amount of marshalling and unmarshalling support for those datatypes internally.

The approach of requiring the application to build messages manually offers applications significant flexibility in message contents while ensuring that the pack and unpack operations are performed by optimized, compiled code. However, relegating message packing and unpacking to the communicating applications means that those applications must have *a priori* agreement on the contents and format of messages. This is not an onerous requirement in small-scale stable systems, but in enterprise-scale distributed computing, the need to simultaneously update all application components in order to change message formats can be a significant impediment to the integration, deployment and evolution of complex systems.

In addition, the semantics of application-side pack/unpack operations generally imply a data copy to or from message buffers. Such copies are known[11, 13] to have a significant impact on communication system performance. Packages which can perform internal marshalling, such as MPI, have an opportunity to avoid data copies and to offer more flexible semantics in matching fields provided by senders and receivers. However, existing packages have failed to capitalize on those opportunities. For example, MPIs type-matching rules require strict *a priori* agreement on the contents of messages. Additionally, most MPI implementations implement marshalling of user-defined datatypes via mechanisms that amount to interpreted versions of field-by-field packing.

This paper describes PBIO(Portable Binary Input/Output)[3], a multi-purpose communication middleware. In developing PBIO we have not attempted to recreate various higher-level communication abstractions offered by MPI or by the Remote Service Requests of Nexus. Instead, we provide flexible heterogeneous binary data transport for simple messaging of a wide range of application data structures, using novel approaches such as dynamic code generation (DCG) to preserve efficiency. In addition, PBIO's flexibility in matching transmitted and expected data

Copyright 2000 IEEE. Published in the Proceedings of the Heterogeneous Computing Workshop, May 1-5, 2000, Cancun, Mexico. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

types provides key support for *application evolution* that is missing from other communication systems.

This paper briefly describes PBIO semantics and features, and then illustrates performance metrics across a heterogeneous environment of Sun Sparc and X86-based machines running Solaris. These metrics are compared against the data communication measurements obtained by using MPI as a data communication mechanism across the same network architecture. The paper will show that the features and flexibility of PBIO do not impose overhead beyond that imposed by other communications systems. In the worst case PBIO performs as well as other systems, and in many cases PBIO offers a significant performance improvement over comparable communications packages.

Much of PBIO's performance advantage is due to its use of dynamic code generation to optimize translations from wire to native format. Because this is a novel feature in communications middleware, its impact on PBIO's performance is also considered independently. In this manner, we show that for purposes of data compatibility, PBIO, along with code generation, can provide reliable, high performance, easy-to-use, easy-to-migrate, heterogeneous support for distributed applications.

2 The PBIO Communication Library

In order to conserve I/O bandwidth and reduce storage and processing requirements, storing and transmitting data in binary form is often desirable. However, transmission of binary data between heterogeneous environments has been problematic. PBIO was developed as a portable self-describing binary data library, providing both stream and file support along with data portability.

The basic approach of the Portable Binary I/O library is straightforward. PBIO is a record-oriented communications medium. Writers of data must provide descriptions of the names, types, sizes and positions of the fields in the records they are writing. Readers must provide similar information for the records they wish to read. No translation is done on the writer's end, our motivation being to offload processing from data providers (e.g., servers) whenever possible. On the reader's end, the format of the incoming record is compared with the format expected by the program. Correspondence between fields in incoming and expected records is established by field name, with no weight placed on size or ordering in the record. If there are discrepancies in field size or placement, then PBIO's conversion routines perform the appropriate translations. Thus, the reader program may read the binary information produced by the writer program despite potential differences in: (1) byte ordering on the reading and writing architectures; (2) differences in sizes of data types (e.g. long and int); and (3) differences in structure layout by compilers.

Since full format information for the incoming record is available prior to reading it, the receiving application can make run-time decisions about the use and processing of incoming messages about whom it had no *a priori* knowledge. However, this additional flexibility comes with the price of potentially complex format conversions on the receiving end. Since the format of incoming records is principally defined by the native formats of the writers and PBIO has no *a priori* knowledge of the native formats used by the program components with which it might communicate, the precise nature of this format conversion must be determined at run-time.

Since high performance applications can ill afford the increased communication costs associated with interpreted format conversion, PBIO uses dynamic code generation to reduce these costs. The customized data conversion routines generated must be able to access and store data elements, convert elements between basic types and call subroutines to convert complex subtypes. Measurements[4] show that the one-time costs of DCG, and the performance gains by then being able to leverage compiled (and compiler-optimized) code, far outweigh the costs of continually interpreting data formats. The analysis in the following section shows that DCG, together with native-format data transmission and copy reduction, allows PBIO to provide its additional type-matching flexibility without negatively impacting performance. In fact, PBIO outperforms our benchmark communications package in all measured situations.

3 Evaluation

In order to thoroughly evaluate PBIO's performance and its utility in high-performance communication, we present a variety of measurements in different circumstances. Where possible, we compare PBIO's performance to the cost of similar operations in MPI. Additionally, we include measurements which evaluate PBIO's performance in situations which are not supported by other communications packages. In particular, we evaluate PBIO's support for application evolution and its ability to transmit dynamically sized data elements.

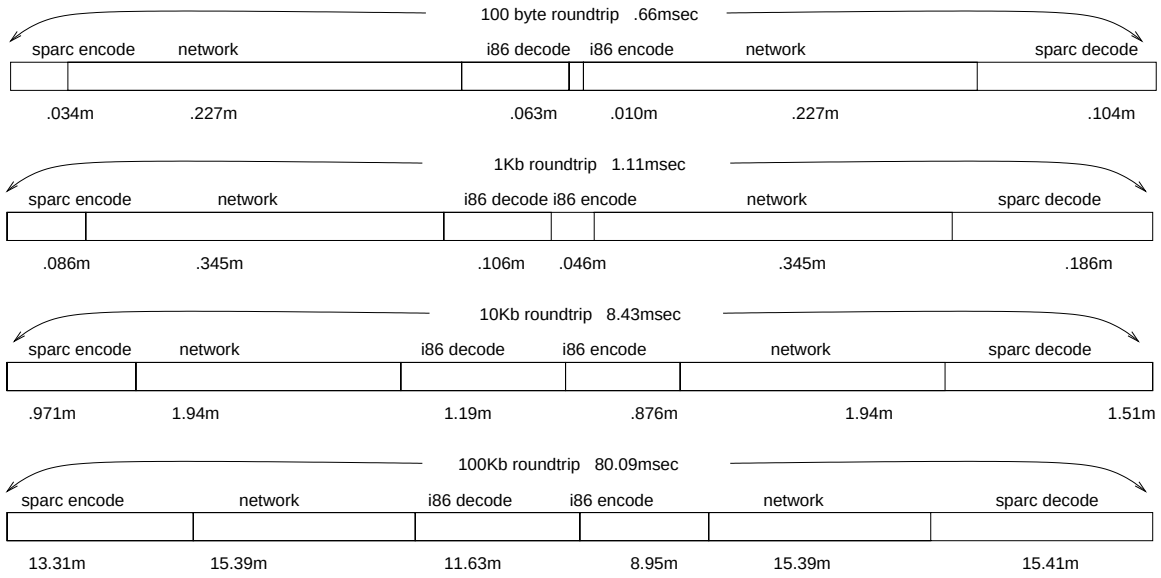

Figure 1: Cost breakdown for message exchange.

3.1 Analysis of costs in heterogeneous data exchange

Before analyzing PBIO costs in detail, it is useful to examine the costs in an exchange of binary data in a heterogeneous environment. As a baseline for this discussion, we use the MPICH[10] implementation of MPI, a popular messaging package in cluster computing environments. Figure 1 represents a breakdown of the costs in an MPI message round-trip between a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[1] The time components labeled "Encode" represent the time span between the application invoking `MPI_send()` and the eventual call to write data on a socket. The "Decode" component is the time span between the recv() call returning and the point at which the data is in a form usable by the application. In generating these numbers network transmission times were measured with NetPerf[9] and send and receive times were measured by substituting dummy calls for socket send() and recv(). This delineation allows us to focus on the encode/decode costs involved in binary data exchange. That these costs are significant is clear from the figure, where they typically represent 66% of the total cost of the exchange.

Figure 1 shows the cost breakdown for messages of a selection of sizes, but in practice, message times depend upon many variables. Some of these variables, such as basic operating system characteristics that affect raw end-to-end TCP/IP performance, are beyond the control of the application or the communication middleware. Different encoding strategies in use by the communication middleware may change the number of raw bytes transmitted over the network, but those differences tend to be negligible. Therefore, the remainder of our analysis will concentrate on the more controllable sending side and receiving side costs.

Another application characteristic which has a strong effect upon end-to-end message exchange time is the precise nature of the data to be sent in the message. It could be a contiguous block of atomic data elements (such as an array of floats), a stride-based element (such as a stripe of a homogeneous array), a structure containing a mix of data elements, or even a complex pointer-based structure. MPI, designed for scientific computing, has strong facilities for homogeneous arrays and strided elements. MPIs support for structures is less efficient than its support for contiguous arrays of atomic data elements, and it doesn't attempt to supported pointer-based structures at all. PBIO doesn't attempt to support strided array access, but otherwise supports all types with equal efficiency, including a non-recursive subset of pointer-based structures. The message type of the 100Kb message in Figure 1 is a non-homogeneous structure taken from the messaging requirements of a real application, a mechanical engineering simula-

[1] The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

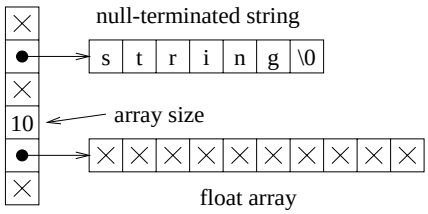

Figure 2: Strings and dynamic arrays in PBIO.

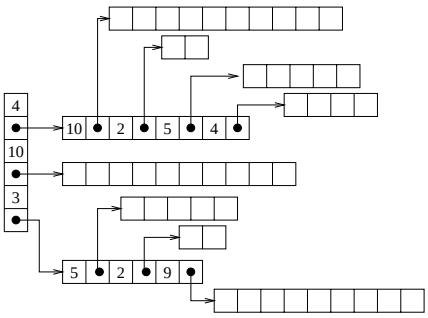

Figure 3: A multi-level pointer structure that can be transmitted by PBIO.

tion of the effects of micro-structural properties on solid-body behavior. The smaller message types are representative subsets of that mixed-type message.

The next sections will examine PBIO's costs in exchanging the same sets of messages. Subsequently, Section 3.5 will examine costs for other data types.

3.2 Sending side cost

As is mentioned in Section 2, PBIO transmits data in the native format of the sender. No copies or data conversions are necessary to prepare simple structure data for transmission. So, while MPICH's costs to prepare for transmission on the Sparc vary from 34μsec for the 100 byte record up to 13 msec for the 100Kb record, PBIO's cost is a flat 3 μsec. Of course, this efficiency is accomplished by moving most of the complexity to the receiver, where Section 3.3 tells a more complex story.

As mentioned above, PBIO also supports the transmission of some pointer-based structures. In particular, PBIO allows an element of a structure be to a null-terminated string, or a pointer to a dynamically sized array,[2] as shown in Figure 2. The array elements may be of an atomic data type or a previously registered structure. That there is no forward declaration mechanism or self-referentiality for structure types restricts PBIO from describing such things as linked lists. However, relatively complex structures, such as the one depicted in Figure 3 can be directly transmitted. The ability to directly transmit dynamically sized arrays is a feature that is not normally present in communications middleware.

Unlike contiguous structures, pointer-based entities do require some preparation before they are sent. In particular, PBIO must walk the structure to 1) prepare a transmission list of data blocks and their lengths, and 2) change all internal pointers from addresses to offsets within the message. The type semantics ensures that there can be no circularities in the structure, so the 'walk' is a simple tree descent which stops when it reaches the 'leaf' structures which contain no pointers. In order to avoid changing the data directly, structures containing pointers are copied to temporary memory and the pointers modified there. This imposes a cost on the sender that is proportional to the amount of data that must be copied and the number of pointers that must be adjusted. Because no similar features are included in common communications libraries, we don't include any representative measurements of these costs. However, we do observe that in the most common use of dynamic arrays, where a relatively small base structure holds pointers and sizes for one or more arrays, the 'walk' is a simple pass over the base structure, the majority of the data is in the 'leaves' which are *not* copied, and the additional sender-side processing is not overly significant.

3.3 Receiving side cost

PBIO's approach to binary data exchange eliminates sender-side processing by transmitting in the sender's native format and isolating the complexity of managing heterogeneity in the receiver. Essentially, the receiver must perform a conversion from the various incoming 'wire' formats to the receiver's 'native' format. PBIO matches fields by name, so a conversion may require byte-order changes (byte-swapping), movement of data from one offset to another, or even a change in the basic size of the data type (for example, from a 4-byte integer to an 8-byte integer).

This conversion is another form of the "marshaling problem" that occurs widely in RPC implementations[1] and in network communication. That marshaling can be a significant overhead is also well known[2, 14], and tools such as USC[12] attempt to optimize marshaling with compile-time solutions.

[2] In the case of a dynamically sized array, the array size must be given by another, integer-typed, element in the base structure.

Unfortunately, the dynamic form of the marshaling problem in PBIO, where the layout and even the complete field contents of the incoming record are unknown until run-time, rules out such static solutions. The conversion overhead is nil for some homogeneous data exchanges, but as Figure 1 shows, the overhead is high (66%) for some heterogeneous exchanges.

Generically, receiver-side overhead in communication middleware has several components which can be traded off against each other to some extent. Those basic costs are:

- byte-order conversion,
- data movement costs, *and*
- control costs.

Byte order conversion costs are to some extent unavoidable. If the communicating machines use different byte orders, the translation must be performed somewhere regardless of the capabilities of the communications package.

Data movement costs are harder to quantify. If byteswapping is necessary, data movement can be performed as part of the process without incurring significant additional costs. Otherwise, clever design of the communications middleware can often avoid copying data. However, packages that define a 'wire' format for transmitted data have a harder time being clever in this area. One of the basic difficulties is that the native format for mixed-datatype structures on most architectures has gaps, unused areas between fields, inserted by the compiler to satisfy data alignment requirements. To avoid making assumptions about the alignment requirements of the machines they run on, most packages use wire formats which are fully packed and have no gaps. This mismatch *forces* a data copy operation in situations where a clever communications system might otherwise have avoided it.

Control costs represent the overhead of iterating through the fields in the record and deciding what to do next. Packages which require the application to marshal and unmarshal their own data have the advantage that this process occurs in special-purpose compiler-optimized code, minimizing control costs. However, to keep that code simple and portable, such systems uniformly rely on communicating in a predefined wire format, incurring the data movement costs described in the previous paragraph.

Packages that marshal data themselves typically use an alternative approach to control, where the marshalling process is controlled by what amounts to a table-driven interpreter. This interpreter marshals or unmarshals application-defined data making

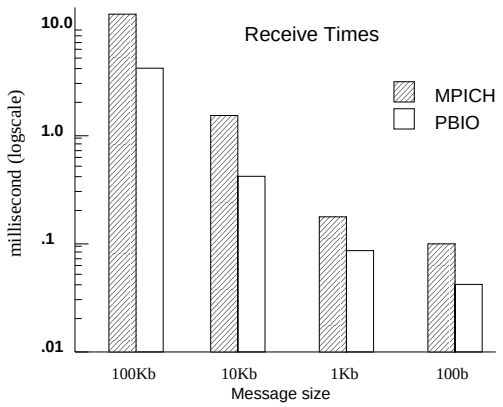

Figure 4: Receiver side costs for PBIO and MPI interpreted conversions.

data movement and conversion decisions based upon a description of the structure provided by the application and its knowledge of the format of the incoming record. This approach to data conversion gives the package significant flexibility in reacting to changes in the incoming data and was our initial choice for PBIO. Figure 4 shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by MPICH (via the `MPI_Unpack()`) call and PBIO. PBIO's converter is relatively heavily optimized and performs considerably better than MPI, in part because MPICH uses a separate buffer for the unpacked message rather than reusing the receive buffer (as PBIO does). However, PBIO's receiver-side conversion costs still contribute roughly 20% of the cost of an end-to-end message exchange. While a portion of this conversion overhead must be the consequence of the raw number of operations involved in performing the data conversion, we believed that a significant fraction of this overhead was due to the fact that the conversion is essentially being performed by an interpreter.

Our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time, making a remedy to the problem of interpretation overhead difficult. However, our solution to the problem was to employ dynamic code generation to create a customized conversion subroutine for every incoming record type. These routines are generated by the receiver on the fly, as soon as the wire format is known, through a procedure that structurally resembles the interpreted conversion itself. However, instead of performing the conversion this procedure directly generates machine code for performing the conversion.

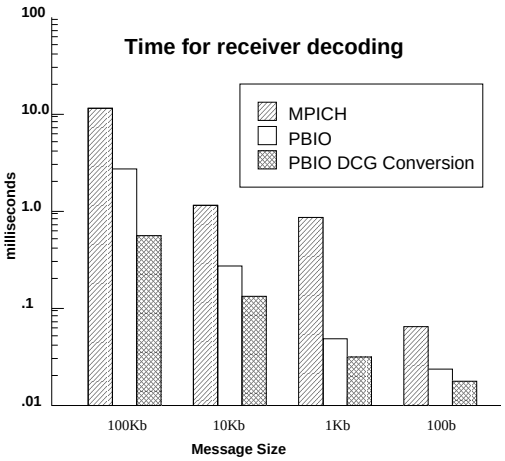

Figure 5: Receiver side costs for interpreted conversions in MPI and PBIO and DCG conversions in PBIO.

The execution times for these dynamically generated conversion routines are shown in Figure 5. The dynamically generated conversion routine operates significantly faster than the interpreted version. This improvement removes conversion as a major cost in communication, bringing it down to near the level of a copy operation, and is the key to PBIO's ability to efficiently perform many of its functions.

The cost savings achieved by PBIO through the techniques described in this section are directly reflected in the time required for an end-to-end message exchange. Figure 6 shows a comparison of PBIO and MPICH message exchange times for mixed-field structures of various sizes. The performance differences are substantial, particularly for large message sizes where PBIO can accomplish a round-trip in 45% of the time required by MPICH. The performance gains are due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*
- using dynamic code generation to customize a conversion routine on the receiving side (currently not done on the x86 side).

3.4 Details of dynamic code generation

The dynamic code generation in PBIO is performed by Vcode, a fast dynamic code generation package developed at MIT by Dawson Engler[5]. We have significantly enhanced Vcode and ported it to several new architectures. The present implementation we can generate code for Sparc (v8, v9 and v9 64-bit), MIPS (old 32-bit, new 32-bit and 64-bit ABIs) and DEC Alpha architectures. An x86 port of Vcode is in progress, but not yet sufficiently advanced for us to generate PBIO's conversion routines. Vcode essentially provides an API for a virtual RISC instruction set. The provided instruction set is relatively generic, so that most Vcode instruction macros generate only one or two native machine instructions. Native machine instructions are generated directly into a memory buffer and can be executed without reference to an external compiler or linker.

Employing DCG for conversions means that PBIO must bear the cost of generating the code as well as executing it. Because the format information in PBIO is transmitted only once on each connection and data tends to be transmitted many times, conversion generation is not normally a significant overhead. Yet that overhead must still be considered to determine whether or not the use of DCG results in performance gains.

The proportional overhead encountered in actually generating conversion code varies dramatically depending upon the internal structure of the record. This differs from the situation in Figure 5, where the worst-case conversion run-time is more dependent upon the size of the message than its structure. To understand this variation, consider the conversion of a record that contains large internal arrays. In this case, the conversion code consists of a few *for* loops that process large amounts of data. In comparison, a record of similar size consisting solely of independent fields of atomic data types requires custom code for each field. The result is that for records consisting solely of arrays, DCG almost always improves performance. For array-based records of around 200 bytes the time to generate and execute dynamic conversion code is less than the time to perform an interpreted conversion. At that point, DCG is a performance improvement, even if the conversion routine is only used once.

The situation is less clear for record formats consisting mostly of individual atomic fields. For this type of record, dynamically generated conversions run nearly an order of magnitude faster than interpreted conversions, but the one-time cost of doing the code generation is relatively high. Obviously, if many records are exchanged, the costs will be amortized over the improved conversion times. But for one-time exchanges dynamic code generation for conversions may be more expensive than simple interpreted conversions.

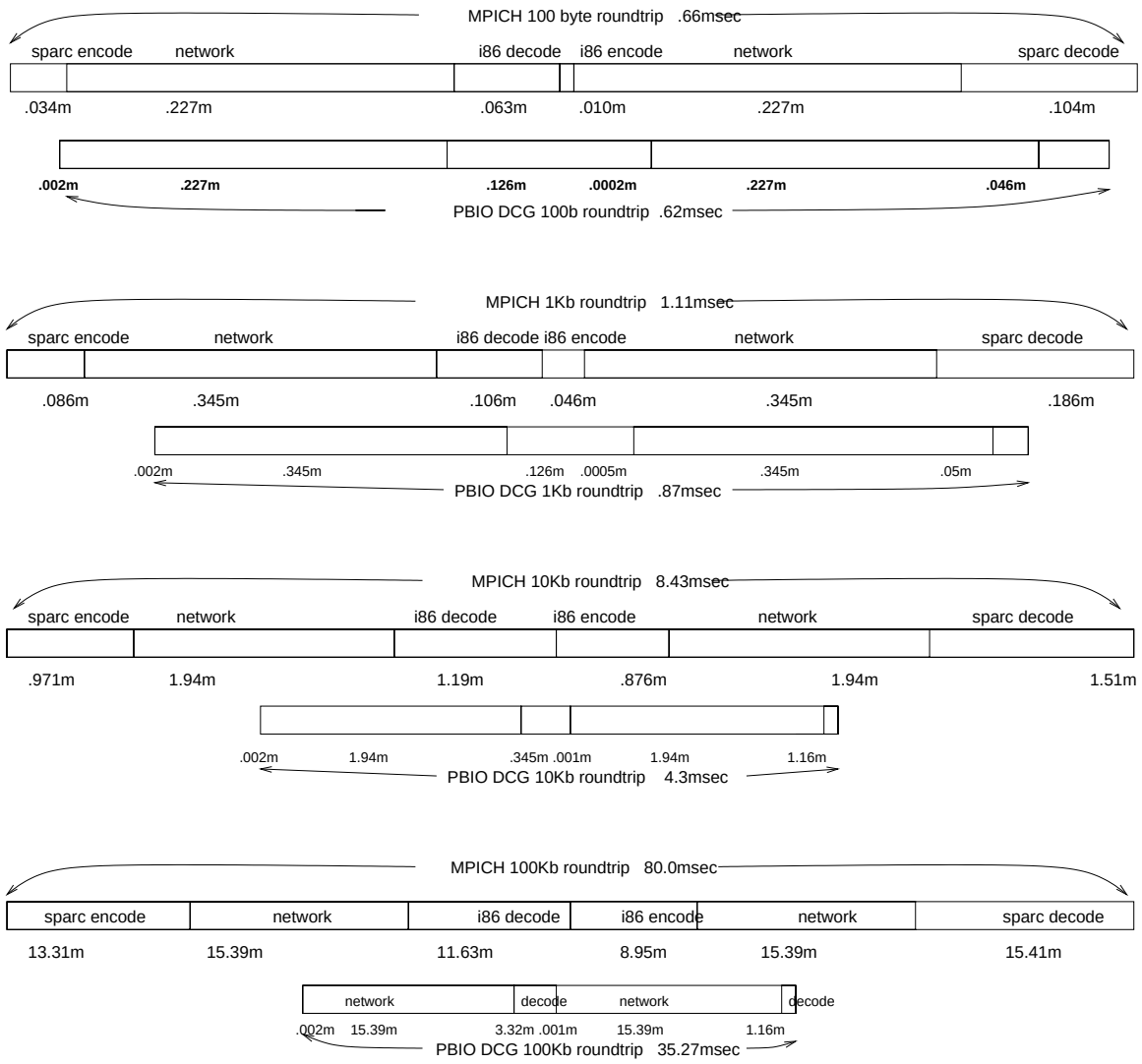

Figure 6: Cost comparison for PBIO and MPICH message exchange.

```
start of proceedure bookkeeping
    save  %sp, -360, %sp
byteswap load and store the 'ivalue' field.
    clr  %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
    st  %g2, [ %i1 ]

byteswap load and store the 'dvalue' field
    mov  4, %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
    mov  8, %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g3
    st  %g3, [ %sp + 0x158 ]
    st  %g2, [ %sp + 0x15c ]
    ldd  [ %sp + 0x158 ], %f4
    std  %f4, [ %i1 + 8 ]

loop to handle 'iarray'
save 'incoming' and 'destination' pointers for later
restoration
    st  %i0, [ %sp + 0x160 ]
    st  %i1, [ %sp + 0x164 ]

make regs i0 and i1 point to start of incoming and
destination float arrays
    add  %i0, 0xc, %i0
    add  %i1, 0x10, %i1
setup loop counter
    mov  5, %g3

loop body.
    clr  %g1
    ldswa  [ %i0 + %g1 ] #ASI_P_L, %g2
    st  %g2, [ %i1 ]

end of loop, increment 'incoming' and 'destination',
decrement loop count, test for end and branch
    dec  %g3
    add  %i0, 4, %i0
    add  %i1, 4, %i1
    cmp  %g3, 0
    bg,a   0x185c70
    clr  %g1
reload original 'incoming' and 'destination' pointers
    ld  [ %sp + 0x160 ], %i0
    ld  [ %sp + 0x164 ], %i1

end-of-procedure bookkeeping
    ret
    restore
```

Figure 7: A sample DCG conversion routine.

For the reader desiring more information on the precise nature of the code that is generated, we include a small sample subroutine in Figure 7. This particular conversion subroutine converts message data received from an x86 machine into native Sparc data. The message being exchange has a relatively simple structure:

```
typedef struct small_record {
    int ivalue;
    double dvalue;
    int iarray[5];
};
```

Since the record is being sent from an x86 and PBIO always sends data in the sender's native data formats and layout, the "wire" and native formats differ in both byte order and alignment. In particular, the floating point value is aligned on a 4-byte boundary in the x86 format and on an 8-byte boundary on the Sparc. The subroutine takes two arguments. The first argument in register `%i0` is a pointer to the incoming "wire format" record. The second argument in register `%i1` is a pointer to the desired destination, where the converted record is to be written in native Sparc format.

The exact details of the code are interesting for a couple of points. First, we make use of the SparcV9 Load from Alternate Space instructions which can perform byteswapping in hardware during the fetch from memory. This yields a significant savings over byteswapping with register shifts and masks. Since this is not an instruction that is normally generated by compilers in any situation, being able to use it directly in this situation is one of the advantages of dynamic code generation.

Second, from an optimization point of view, the generated code is actually quite poor. Among other things, it performs two instructions when one would obviously suffice, and unnecessarily generates an extra load/store pair to get the double value into a float register. The are several reasons for this suboptimal code generation, including the generic nature of the virtual RISC instruction set offered by Vcode, the lack of an optimizer to repair it, and the fact that we have not seriously attempted to make the code generation better. Even when generating poor code, DCG conversions are a significant improvement over other approaches.

Examining the generated code may also bring to mind another lurking subtlety in generating conversion routines: data alignment. The alignment of fields in the incoming record reflects the restrictions of the sender. If the receiver has more stringent re-

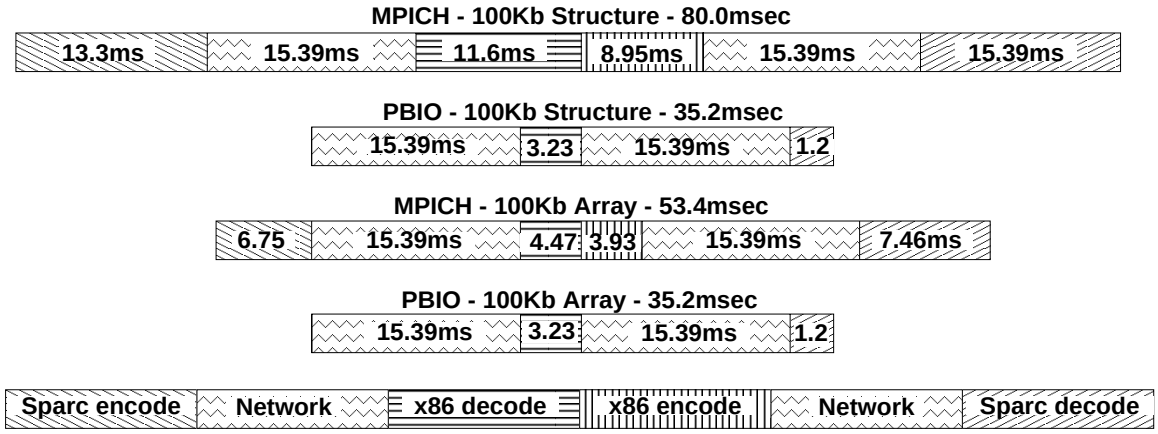

Figure 8: Comparison between PBIO and MPICH in structure and array exchange time.

	MPICH			PBIO		
data size	total time	send side overhead	receive side overhead	total time	send side overhead	receive side overhead
100Kb	20.8ms	0.46	0.78	18.3	0.0028	0.034
10Kb	3.02ms	0.083	0.20	2.52	0.0028	0.034
1Kb	1.06ms	0.0097	0.086	0.90	0.0028	0.034
100b	.63ms	0.0056	0.076	0.52	0.0028	0.034

Table 1: A comparison of PBIO and MPICH for homogeneous exchange of arrays

strictions, the generated load instruction may end up referencing a misaligned address, a fatal error on many architectures. This situation would actually have occurred in in the example shown in Figure 7, where the incoming `double` array is aligned on a 4 byte boundary because the Sparc requires 8 byte alignment for 8-byte loads. Fortunately, the suboptimal Sparc dynamic code generator loads the two halves of the incoming 8-byte doubles with separate `ldswa` instructions instead of a single `lddfa` instruction.

Data alignment is generally not an issue in storing to the native record because it is presumably aligned according to the requirements of the receiving machine. We also assume that the base addresses of the incoming and native records are strongly aligned. This leaves the offsets of the incoming record fields as the primary source of misalignment. Since these are known at code generation time, we can make static decisions about using efficient direct loads for aligned data or using potentially less efficient methods for unaligned data.[3]

[3]Our current code generator does not handle misaligned accesses, but the extension to handle them is straightforward.

3.5 Other data types and homogeneous systems

The previous sections compared PBIO's performance with that of MPICH in situations involving a heterogeneous exchange of structures containing mixed types. While PBIO shows clear and significant performance gains over MPICH in that situation, MPICH might be expected to perform better in dealing with messages consisting of contiguous arrays, or in a homogeneous exchange where it might not use an XDR-based encoding scheme.

Figure 8 shows a breakdown of MPICH and PBIO performance for heterogeneous transmission of a 100Kb floating point array and compares it to the previously presented breakdowns for the 100Kb structure. This figure shows that PBIO's performance remains essentially unchanged when the datatype is changed from structure to an array. MPICH performance does improve with contiguous arrays, but not to the point where it matches PBIO's performance. The results for smaller datatypes are similar.

A comparison of round-trip times for contiguous arrays between homogenous machines is shown in Table 1. This is one of the simplest cases in binary

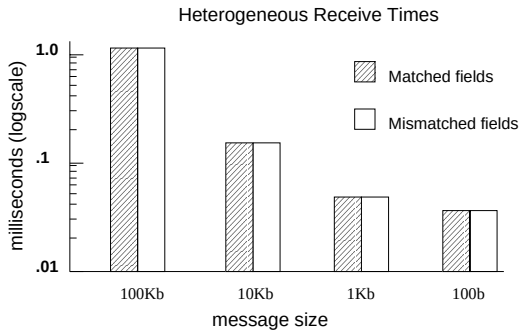

Figure 9: Receiver-side decoding costs with and without an unexpected field – heterogeneous case.

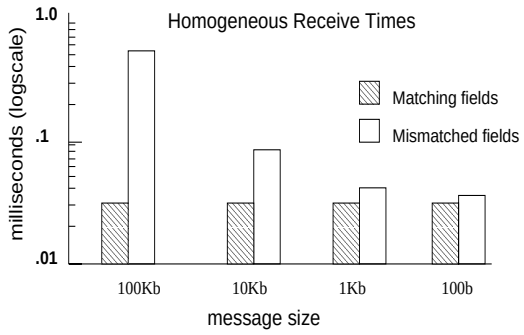

Figure 10: Receiver-side decoding costs with and without an unexpected field – homogeneous case.

communication, requiring no data conversion of any kind. The send and receive side overheads are tiny compared to the time required for network transmission, but PBIO retains a slight edge over MPICH in both receive and send side overheads. These differences largely account for the 10% or so better performance that PBIO achieves in round-trip time for these contiguous arrays.

That PBIO has better performance than MPICH even in situations where MPICH might be expected to prevail is convincing evidence that PBIO's extra flexibility in supporting application evolution does not negatively impact performance in other situations. The next section will examine PBIO's performance in the presence of application evolution.

3.6 Performance in application evolution

The principal difference between PBIO and most other messaging middleware is that PBIO messages carry format meta-information, somewhat like an XML-style description of the message content. This meta-information can be an incredibly useful tool in building and deploying enterprise-level distributed systems because it 1) allows generic components to operate upon data about which they have no *a priori* knowledge, and 2) allows the evolution and extension of the basic message formats used by an application without requiring simultaneous upgrades to all application components. In other terms, PBIO allows *reflection* and *type extension*. Both of these are valuable features commonly associated with object systems.

PBIO supports reflection by allowing message formats to be inspected before the message is received. It's support of type extension derives from doing field matching between incoming and expected records by name. Because of this, new fields can be added to messages without disruption because application components which don't expect the new fields will simply ignore them.

Most systems which support reflection and type extension in messaging, such as systems which use XML as a wire format or which marshal objects as messages, suffer prohibitively poor performance compared to systems such as MPI which have no such support. Therefore, it is interesting to examine the effect of exploiting these features upon PBIO performance. In particular, we measure the performance effect of type extension by introducing an unexpected field into the incoming message and measuring the change in receiver-side processing.

Figures 9 and 10 present receive-side processing costs for an exchange of data with an unexpected field. These figures show values measured on the Sparc side of heterogeneous and homogeneous exchanges, respectively, using PBIO's dynamic code generation facilities to create conversion routines. It's clear from Figure 9 that the extra field has no effect upon the receive-side performance. Transmitting would have added slightly to the network transmission time, but otherwise the support of type extension adds no cost to this exchange.

Figure 10 shows the effect of the presence of an unexpected field in the homogeneous case. Here, the overhead is potentially significant because the homogeneous case normally imposes no conversion overhead in PBIO. The presence of the unexpected field creates a layout mismatch between the wire and native record formats and as a result the conversion routine must relocate the fields. As the figure shows, the resulting overhead is non-negligible, but not as high as exists in the heterogeneous case. For smaller record sizes, most of the cost of receiving data is ac-

tually caused by the overhead of the kernel `select()` call. The difference between the overheads for matching and extra field cases is roughly comparable to the cost of `memcpy()` operation for the same amount of data.

The results shown in Figure 10 are actually based upon a worst-case assumption, where an unexpected field appears before all expected fields in the record, causing field offset mismatches in all expected fields. In general, the overhead imposed by a mismatch varies proportionally with the extent of the mismatch. An evolving application might exploit this feature of PBIO by adding any additional at the end of existing record formats. This would minimize the overhead caused to application components which have not been updated.

4 Conclusions

Current distributed applications rely heavily on leveraging the computing power of heterogeneous networks of computer architectures. The PBIO library is a valuable addition to the mechanisms available for handling binary data interchange among these heterogeneous distributed systems. PBIO performs efficient data translations, and supports simple, transparent system evolution of distributed applications, both on a software and a hardware basis.

Rather than relegating message packing and unpacking operations to the communicating applications, thus requiring *a priori* agreement on these data structures, PBIO efficiently layers and abstracts diversities in computer architectures. Applications need only agree on data by *name*, and previously exposed concerns such as byte ordering, architecture specifications, data type sizes, and compiler differences are no longer a concern. Since PBIO uses dynamic code generation rather than data interpretation, compiler optimizations are utilized without the cumbersome limitations of static data structures.

Enterprise-scale distributed computing can be implemented and deployed much more simply and efficiently using PBIO's flexibility, not only initially, but during the evolution of specific distributed components. Data elements can be incrementally to the basic message formats of distributed applications without disrupting the operation of existing application components.

The measurements in this paper have shown that PBIO's flexibility does not impact its performance. In fact, PBIO's performance is better than that of a popular MPI implementation in every test case, and significantly better in heterogeneous exchanges. Performance gains of up to 60% are largely due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*
- using dynamic code generation to perform data conversion on the receiving side.

In short, PBIO is a novel messaging middleware that combines significant flexibility improvements with an efficient implementation to offer distributed applications fast heterogeneous binary data interchange.

References

[1] Guy T. Almes. The impact of language and system on remote procedure call design. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 414–421. IEEE, May 1986.

[2] D.D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Synposium*, pages 200–208, Sept 1990.

[3] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu).*

[4] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, 24(12-13), 1998.

[5] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[6] Message Passing Interface (MPI) Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[7] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[9] Hewlet-Packard. The netperf network performance benchmark. http://www.netperf.org.

[10] Argonne National Laboratory. Mpich-a portable implementation of mpi. http://www-unix.mcs.anl.gov/mpi/mpich.

[11] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7)*, July 1998.

[12] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[13] Marcel-Catalin Rosu, Karsten Schwan, and Richard Fujimoto. Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture. *Cluster Computing, Special Issue on High Performance Distributed Computing*, 1, January 1998.

[14] M. Schroeder and M. Burrows. Performance or firefly rpc. In *Twelfth ACM Symposium on Operating Systems, SIGOPS, 23, 5*, pages 83–90. ACM, SIGOPS, Dec. 1989.

Greg Eisenhauer is a research scientist in the College of Computing at the Georgia Institute of Technology. He received his PhD from Georgia Tech in 1998 under the direction of Dr. Karsten Schwan. Dr. Eisenhauer previously worked at Honeywell's Systems and Research Center and received his BS and MS degrees from the University of Illinois, Champaign-Urbana. His research interests include interactive computational steering, performance evaluation and scientific computing.

Lynn K. Daley is a PhD student in the College of Computing at Georgia Institute of Technology, working under the direction of Dr. Karsten Schwan. Ms. Daley has over 20 years software development experience, having worked at Harris Government Systems, Digital Equipment Corp., and Atlanta Signal Processors. She holds BS/CS and MS/EE degrees from Georgia Institute of Tech, and a MS/EngMgmt from Florida Institute of Technology. Her research interests include parallel, distributed, and real-time processing.