

XBS: A Streaming Binary Serializer for High Performance Computing

Kenneth Chiu

Indiana University
chiuk@cs.indiana.edu

Abstract

High performance distributed systems communication requires that data first be serialized into a byte sequence suitable for transmission. A variety of different formats exist for serialization, ranging from XML-based formats to more efficient binary formats. This paper presents the XBS binary serialization library. XBS differs from other binary serializers in that it is a streaming serializer (as opposed to message-oriented), and that it is object-oriented. Streaming serializers are generally more flexible, and we show that they can also be more efficient in some situations. XBS is written in C++, and its object-orientation facilitates its use in modern high performance applications.

1 Introduction

Addressing high performance computing challenges increasingly requires large-scale distributed systems. In some situations, such as Grid computing, the interacting processes are Grid services, which are loosely coupled and may be widely dispersed. In others, such as supercomputer clusters, the interactions are between tightly coupled computations.

In each of these cases, the interactions comprise exchanges of data. In the Grid computing case, the data might be the results of a computation being sent to a remote site for visualization. In the cluster case, the data might be a time-step update.

The data takes different forms during the interaction. When active in a running program, its form is dictated by the hardware architecture and the programming language. For structured data, this form may be highly complex and discontinuous. To transmit it, or store it in a file, however, the data must usually be converted to a byte sequence of some kind. This process is known as *serialization*, and can be a significant contributor to communication costs.

Data can be serialized into a variety of formats. Some formats, such as textual ones, are more appropriate for interoperability and long term archiving. Others can offer much higher performance, and are more suited for the exchange of transient data during a computation.

In this paper we examine a class of serializers known as binary serializers. Binary serializers use non-textual formats, and represent numbers in a form closer to that required by computing hardware. The cost of textual formats is often not so much in the inherent verbosity of text, but the need to convert from a decimal representation to a binary one.

A binary serializer can be implemented in a number of different ways, and a number of different standards and

implementations have been developed. Different design decisions affect the suitability to different use cases.

In this paper we present the XBS serialization library. XBS differs significantly from other binary serializers in that it is stream-based and written in object-oriented C++. Contributions include

- An object-oriented, C++ application interface.
- Use of templates and polymorphism for flexible and efficient handling of the serialized data stream.
- A streaming programming model that avoids the inefficiencies of message-based models in high-rate output situations.

2 Goals

A binary serializer can have many different designs and implementations, reflecting the intended use cases. Some designs will be more appropriate in some situations than others. We foresee the XBS serializer being used in situations where good performance is a priority, but perhaps not of utmost importance. An ad hoc, hand-coded serializer will be more efficient, as our results show. Equally important for us is integration into large-scale, middleware systems.

We also see XBS used in situations where it is necessary to transfer large, complex, and dynamic data structures, as opposed to small, occasional events. In such situations, serializers based strictly on messages suffer.

We detail the goals below.

2.1 Lightweight

XBS is designed as a low-level serialization library. It provides only the functionality to convert in-memory native representations of numbers and strings to byte sequences. It does not impose any kind higher-level structure on the data, such as some kind of record-orientation.

The motivation for this is to foster a flexible, layered approach. This allows XBS to be used in many different higher-level protocols. For example, this approach allows XBS to be used to define a “binary XML” format. In addition to defining an actual, text-based syntax, XML also implicitly defines an abstract model known as an XML Infoset. This abstract model can be mapped to a variety of actual formats, including XBS. This would provide improved performance due to the obviation of text processing.

XBS also does not impose any kind of self-description in the data stream. This means that applications that send pre-defined sequences of data (implicit typing) do not incur the additional overhead of inserting type information.

On the other hand, applications that need some degree of explicit typing can easily insert their own typecodes. (In fact XBS does this when compiled with debugging enabled.)

2.2 Disposition Independence

The process of moving data from its programmatic representation in memory to some other medium logically involves two steps. The first step is serialization. The second step is to transfer this serialized representation to some medium, such as disk or network. We term this step the *disposition* of the serialized data.

The serialization library should maintain a clear separation between these two steps, and provide a flexible interface to the actual device-specific mechanisms. XBS attempts to allow arbitrary actual transport mechanisms, such as TCP/IP or perhaps something Myrinet-related.

2.3 Streaming

Some serialization libraries group data into fixed message units. Each message is sent or received atomically, and the contents of the message are fixed at compile time. Such libraries can provide convenient means to organize the transfer of information, but can also be unnecessarily restrictive.

For some applications, the data to be streamed is highly dynamic in nature. For example, a complex mesh structure simply cannot be placed into a fixed message, since its exact structure is not known at compile time. It is still possible to simply use a dynamic sequence of small messages, but then performance suffers, since these libraries are typically optimized for larger messages.

Message-oriented libraries can also be limiting if the message structure does not correspond to the application data structures. In these cases, the application data must first be copied into the message before it can be transmitted.

For these reasons, we chose a streaming API for XBS. No message structure is imposed. The data is simply streamed as a sequence of numbers and strings. This allows the stream to be freely and dynamically structured based on the run-time data structures.

Note that a streaming interface does not preclude the organization of the data stream into messages, or even more sophisticated hierarchical structures such as those corresponding to various object models like C++ classes.

2.4 C++ Interface

Modern, high performance distributed systems are often implemented in C++. A C++-based serializer facilitates integration into these systems.

As detailed in Section 4.2, we also use C++ features to assist in the attainment of our other goals.

2.5 Performance

Performance is of course an important, though somewhat subjective, goal. The features described above should not impose too high of a cost.

3 Performance Issues

In this section we examine performance more closely. Attaining high performance depends on satisfying a number of competing requirements.

- Minimize the cost per datum.
- Minimize the total number of system calls.
- Minimize memory costs such as paging and cache misses.
- Maximize I/O-computation overlap.

Simultaneously satisfying all of these is generally not possible, as detailed below. This suggests that an ideal system would provide a number of performance tuning parameters, along with a parameter-space search tool.

3.1 Minimize Cost Per Datum

To minimize this, the wire format should be reasonably close to the CPU representation. A large majority of today's high performance hardware supports two's-complement integers and IEEE 754 floating-point numbers, including Itanium, Pentium, IBM POWER, MIPS, Alpha, Sun SPARC, Cray X1, and the Earth Simulator. Thus, two's complement and IEEE 754 are a good choice.

The data is moved into a buffer before disposition. Using an appropriately-typed assignment statement is significantly more efficient than using a call to a memory copying routine such as `memcpy()`.

```
char *buf = ...;
double x = ...;
*(double *)buf = x; // Assume proper alignment.
```

Inlined functions can also enhance performance. C++ and the new C99 standard support inlining.

If buffering is used, the capacity of the buffer is typically tested after every datum. If the buffer is full, it is flushed. This if-test is usually acceptable, but can incur a penalty in some situations.

3.2 Minimize System Calls

Initiating a network transmission generally requires executing a sequence of software and hardware steps. Once a transmission has been initiated, however, the marginal cost of each successive byte is usually a small fraction of the overhead. Thus, the number of system calls should be minimized,

and the amount of work done in each system call should be maximized.

3.3 Minimize Cache Misses

The requirement to minimize the number of system calls suggests that a maximal amount of work should be done in each call. This in turn would require the use of a buffer large enough for the entire dataset. Large buffers tend to incur memory costs such as cache misses and paging, however. The problem when using a large buffer is that by the time the end of the buffer has been reached, the beginning of the buffer has already been evicted from cache. Any succeeding processing must access the bytes from main memory rather than cache.

On some hardware direct memory access (DMA) is used to transfer data directly from main memory to the hardware. This could in theory preclude any benefits of attempting to keep data in cache, since the DMA would bypass the cache. However, generally speaking DMA can only access kernel memory, not user memory. Thus, the data must be transferred from user memory to kernel memory before the DMA can be initiated. This transfer is faster if the data is still in cache, so minimizing cache misses is still beneficial.

On a few systems, true zero-copy is possible. In this case, the data is transferred directly from user memory to the network. In such situations, minimizing cache misses is not as useful.

3.4 Maximize I/O-Computation Overlap

Many operating systems perform the I/O in the background after an I/O system call returns. This improves performance, since the CPU can perform computations while the I/O is occurring. I/O is often not CPU-intensive, so there is often CPU headroom available for computations during I/O. Excessive buffering, however, can minimize the amount of overlap that occurs. If the buffer is very large, there may be no computation to overlap with the I/O, as can be seen in Figure 1. Achieving overlap requires careful design of the coupling between the computational and communication aspects of an application.

4 The XBS Library

In this section we describe the XBS library. A binary serializer implementation can define two orthogonal aspects: format and programming interface (API). Though separate in theory, these are often conflated in practice, and XBS is no exception. We first discuss the data format.

4.1 Data Format

The XBS data format is relatively simple, and is designed to work well with modern architectures. Note that designing a format that will work well with any conceivable hardware

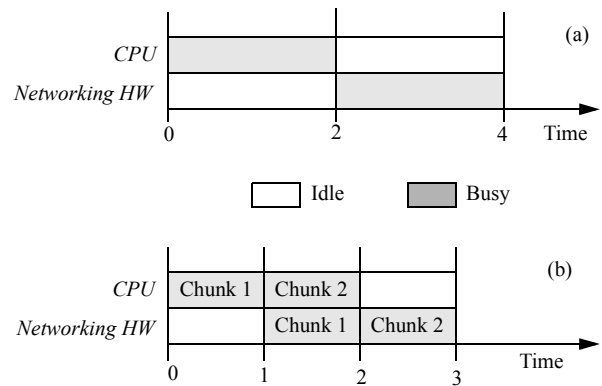


Figure 1. This diagram shows how smaller buffers can increase the overlap between I/O and computation. At time 0, an event occurs which results in a network transmission. We assume that the transmission requires 2 time units of CPU processing and 2 time units of network transmission. In (a), the transmission is wholly buffered before being transmitted. The total time is 4. In (b), however, the data is divided into two chunks. At time 1, the first chunk has been processed by the CPU, and is handed off to the networking hardware for transmission. While the network is transmitting the data, the CPU continues working on the second chunk. The total time is now reduced to 3.

architecture is simply unrealistic. For example, an architecture that does not use two's-complement arithmetic will necessarily entail some conversion costs.

XBS assumes that the machine uses 8-bit bytes. XBS makes no assumptions about the byte-order, but provides a hook in the form of a template parameter to manipulate the byte-order upon reception. For flexibility, actual handling of the byte-order is left to the user of XBS.

As a convenience, the library provides a class to reverse the byte-order. This handles the common case of little- and big-endian architectures.

Note that any architecture that does not conform can still use XBS, as long as it provides a class that can perform the conversion.

XBS is not a self-describing format, but self-description can easily be layered on top. For example, each datum can be preceded by a 1-byte type code. This allows self-description when necessary, but does not impose the penalty when both the sender and receiver have complete knowledge of the data types in the stream.

In fact, when compiled in DEBUG mode, XBS will inject type codes into the stream to provide type safety. This prevents an integer from being mistakenly deserialized as a float, for example. Figure 2 provides an overview of the XBS format

4.1.1 Integers

XBS specifies that integers are in two's complement form. XBS supports 1-, 2-, 4-, and 8-byte integers. Integers are always aligned to a multiple of their size. The base of the alignment is the beginning of the stream, so as long as the

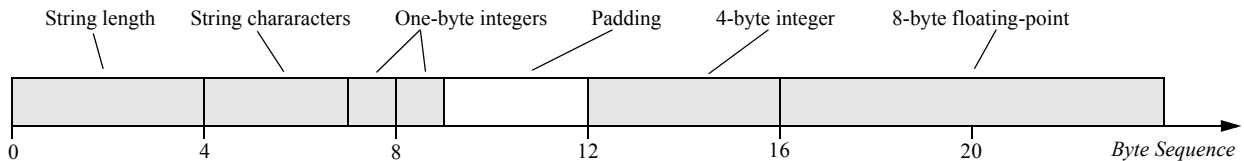


Figure 2. The diagram above indicates the alignment used by the XBS format. The general principle is that all numbers are aligned to their size, but are packed as tightly as possible within this requirement. Thus, an 8-byte floating-point number is aligned to 8, etc. Strings are composed of a 4-byte length followed by the characters.

network buffer is aligned to 8-bytes, the integer will be properly aligned when received or transmitted. This means that an integer can be moved directly into or out of a buffer by a native move instruction of the appropriate length.

4.1.2 Floating-Point

XBS specifies that floating-point numbers are in IEEE 754. As we do for integers, we assume that floating numbers are aligned to a multiple of their size. Thus, an 8-byte double-precision number is aligned to a multiple of 8, etc.

4.1.3 Strings

Strings are in the form of an 32-bit length followed by a sequence of one-byte integers representing the ASCII (or UTF-8) characters. For convenience, the final integer is always 0, which makes strings in buffers compatible with C strings. The length includes the final 0 byte.

As with all numbers, the preceding length is aligned to its size, which is 4 bytes. The following one-byte integers are not further aligned, which means that there may be padding before the length, but that there is never padding between the length and the beginning of the characters in the string.

4.2 Implementation

An ideal software design provides the specified functionality without imposing restrictions on aspects not within its core purpose. XBS concentrates on alignment and buffering. The actual disposition of a buffer once filled is considered the responsibility of the back-end developer, and outside of XBS proper.

Programming languages differ on how they provide a separation of concerns. C provides function pointers and macros. Other languages, such as AspectJ[3], formalize the notion of aspects, and provide much more support for them. C++ provides two primary means of allowing variation in different aspects.

Polymorphism. Polymorphism is obtained in C++ by the use of virtual functions. A derived class inherits from a base class, and provides its own implementation of virtual functions. Thus, the concerns of the virtual function can be separated from the base class.

Class templates. A separation of concerns can also be achieved through the use of class templates. A class template can factor out an aspect into a template parameter. Templates provide a somewhat more flexible way to separate concerns, especially when combined with advanced generic programming techniques. These techniques essentially exploit templates as a metalanguage within C++.

XBS attempts to provide a core serialization functionality without imposing restrictions on other aspects of data transfer, and uses both of these techniques in different places.

XBS uses two parallel sets of interfaces, one for serialization and one for deserialization. The serialization set consists of a `Serializer` class and a `SerializationBuffer` class, and the deserialization set consists of a `Deserializer` class and a `DeserializationBuffer` class. The `Serializer` class is a templated class. The `SerializationBuffer` and `DeserializationBuffer` classes are abstract classes that require a derived class to implement the `fill` and `flush` methods described below.

Though the XBS data format itself is not tied to C++, part of the contribution of this work is to investigate the use of modern C++ features for serialization. The API could be ported to Java, but would lose some of the performance benefits, such as the inlining afforded by the use of templates. A FORTRAN API could also be implemented, but a more practical scenario might be to implement application-specific C routines that would be called from FORTRAN. The C routines would then call XBS C++ code.

4.2.1 Byte Ordering

Different machine architectures may use different orderings for the bytes that compose a multi-byte unit, such as a 64-bit floating-point number. Thus, if a number is read by a machine with a different ordering from the machine that wrote the number, the bytes will need to be reordered.

A serialization library could simply internally support both of the two most common orderings, little-endian and big-endian. XBS instead provides a template parameter that is used for byte-swapping. By using a template parameter, the actual byte-swapping code can be inlined.

4.2.2 Type Coding

A common bug during development is to serialize a datum as one type, but deserialize it as a different type. The result is some garbage value. These bugs are typically time consuming to find.

To reduce the effort of debugging these problems, XBS provides a means to inject type codes into the data stream. These type codes can be used throw a runtime exception if the data is deserialized with the wrong type.

The type coding is implemented as template parameter to the `Serializer` and `Deserializer` classes. The actual argument for this parameter should be a class with two static member templates.

```
template <typename T>
static void
packType(SerializationBuffer *);

template <typename T>
static void
unpackType(DeserializationBuffer *);
```

By implementing these as template parameters to the `Serializer` and `Deserializer` classes, these static member functions can be inlined for efficiency. If they had been implemented as virtual functions, they would necessarily have not been inlined, since the common functionality would have been in a base class.

Two type coding classes are provided for convenience: `TypeCodingOn` and `TypeCodingOff`.

4.2.3 Buffering

The `Serializer` class relies on a corresponding `SerializationBuffer` class to handle the actual disposition of the byte sequence resulting from serialization. The `SerializationBuffer` class has a pure virtual `flush` function, which serves to abstract platform and hardware dependencies. An actual application implements its own `flush` function in a class derived from `SerializationBuffer`. For example, an implementation that uses the standard UNIX file descriptor mechanisms might simply use the `write` system call when `flush` is called. An implementation that uses Fibre Channel might use different system calls.

The `Deserializer` class has a corresponding `DeserializationBuffer` class analogous to the serialization case. The `DeserializationBuffer` class has a virtual `fill` function that is implemented by the application.

The goal of buffering is to minimize the number of system calls by maximizing the amount of work done in each call. There is usually a fair degree of overhead associated with each system call.

A straightforward specification of the `flush` function would simply state the implementation of the `flush` function flushes the buffer.

Unfortunately, this simplicity can increase the number of system calls. The reason is that common operating systems such as UNIX do not guarantee that a write system call will actually write all of the requested bytes. The call may only write a prefix of the request. Thus, if the `flush` function must flush the entire buffer, it would require two system calls.

Instead of mandating that `flush` operation flush the entire buffer, the implementation is called with a minimum flush requirement that is never more than 8 bytes. The `flush` operation provided by the implementation is then free to flush whatever amount is most efficient, as long as it meets the minimum flush.

4.2.4 User Interface

Creating an XBS serializer requires creating a buffer object, and then supplying two template parameters.

```
StreamSerializationBuf buf;
Serializer<ByteSwappingOn, TypeCheckingOn>
s(buf);
```

The `StreamSerializationBuf` class derives from the `SerializationBuffer` class, and is responsible for the actual implementation of the `fill` and `flush` routines. To actually serialize a number, the user calls the `pack` functions.

```
s.pack(1235);
s.pack(1.234);
s.flush();
```

The `flush` call is only necessary at some kind of synchronization point, such as when waiting for a reply. Otherwise, XBS will flush the buffer when full.

The deserializer interface is analogous.

5 Related work

Most computer systems use ad hoc serialization in several places. For example, the logical structures of a file system are represented in serialized form in the actual disk blocks.

The layout, format, and APIs are hand-crafted for the needs at hand, and are often not documented. Because these have limited applicability, we do not consider them further.

5.1 External Data Representation (XDR)

The External Data Representation (XDR) [6] is a standard for the description and encoding of data that was developed by Sun, and is used for the Open Network Computing (ONC) Remote Procedure Call (RPC) [7] protocol. Its most widespread application is the Network File System (NFS).

XDR includes integer, floating-point, enumeration, string, and structure types. XDR requires all data be aligned

to 4-byte words. Eight-byte numbers, such as double-precision floating-point numbers are not required to be aligned to eight bytes, which is problematic for many 64-bit architectures.

XDR specifies little-endian as the canonical byte order, which can result in unnecessary byte swapping when both the sender and receiver use big-endian. Like XBS, XDR is a stream-based serializer.

5.2 Common Data Representation (CDR)

The Object Management Group (OMG) developed the Common Data Representation (CDR) as part of the Common Object Request Broker Architecture (CORBA) [5]. It is a bicononical format, big-endian and little-endian. The writer sends in its natural format, and the reader makes right. It includes 8-byte types, and aligns everything to its size.

5.3 Network Data Representation (NDR)

The Open Software Foundation (now the Open Group) developed the Network Data Representation (NDR). NDR is a multicononical format. Floating-point numbers can be in a number of different formats, including IEEE, VAX, IBM, and Cray.

A variant of NDR is used for DCOM.

5.4 Nexus

Nexus [4] was developed as part of the Globus project, and was widely used. Lately Globus has deprecated Nexus in favor of Web services and SOAP. Nexus is a hybrid between an RMI system and a serializer.

5.5 Portal Binary I/O (PBIO)

PBIO [1] is a high-performance package developed at Georgia Tech. It utilizes dynamic code generation to generate high-performance machine-level packing and unpacking code.

PBIO is message-based, and so may not be suitable for some situations. The message is defined by a C struct, which means that it is likely the message data will need to first read from some other data structure, and then written into the struct, which may be inefficient. Note that because static field offsets are used to access fields in the struct, a C++ class cannot be used unless it is a “plain old data” (POD) class.

5.6 BinX

BinX [8] is a XML language for describing binary datasets. It is capable of describing any binary data that can be encoded in XDR. BinX could also be used to describe XBS streams, though the block size is not needed for XBS since XBS aligns each datum to its individual size, rather than using a constant block size for all data.

Though the XDR format is not compatible with XBS, it is likely that BinX could also be used to describe As currently defined, it cannot be used to describe XBS streams, because XDR requires all data be aligned to 4-byte words,

6 Performance Results

We compared the performance of XBS against four other binary serializers: Nexus, PBIO, XDR, and a hand-coded raw serializer. We tested message sizes from 1 double to 2048 doubles. We assume that the scenario is a high-throughput streaming one, so we do not flush the XBS buffer after every message.

The Nexus, PBIO, and XDR tests used the default transport for the libraries. The XBS and hand-coded tests were implemented with UNIX sockets over TCP/IP.

The Linux machines were 2.8 GHz Pentium Xeons. The Solaris machines were Sun Fire 280R equipped with 1.2 GHz SPARC III CPUs. All machines were interconnected via switched Gigabit Ethernet.

As can be seen in Figure 3, XBS performs equally well in most situations. When writing from Linux to Solaris, the raw serializer is significantly better, however. For small messages XBS performs better than the raw serializer because the raw serializer does not aggregate small messages into one write.

In most cases, XBS performs better due to buffering. Libraries such as PBIO also perform buffering, but the granularity is restricted to that of a single message. Thus, if the application requires sending a series of small messages, XBS can be significantly faster. If the sequence is known at compile-time, then the sequence can be merged into one message, and PBIO can obtain high performance. If, however, the sequence is not known till run-time, then PBIO cannot merge the messages into one message.

XDR buffers similarly to XBS, but does not perform as well, for unclear reasons. We surmise that it is due to a combination of lack of inlining, and byte swapping. XBS also byte swaps, but the byte swapping is inlined.

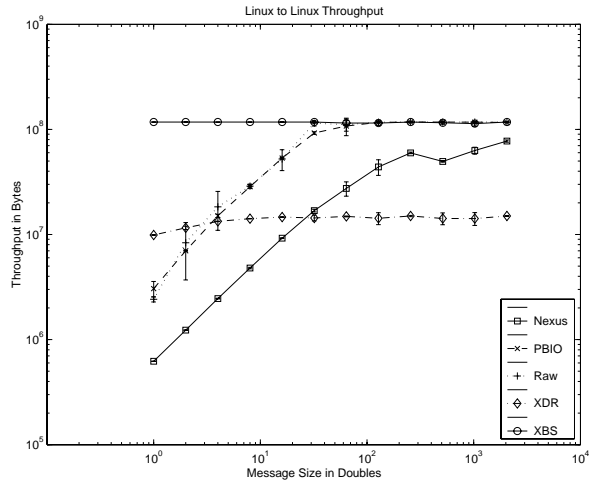
7 Conclusion

XBS is high-performance, lightweight, streaming binary serializer. It sparingly utilizes some advanced C++ features to separate the disposition of the data from the narrower job of alignment and buffering. The use of a template parameter allows a flexible byte-swapping implementation that can still be inlined, and the injection of type codes for self-describing streams (explicit typing).

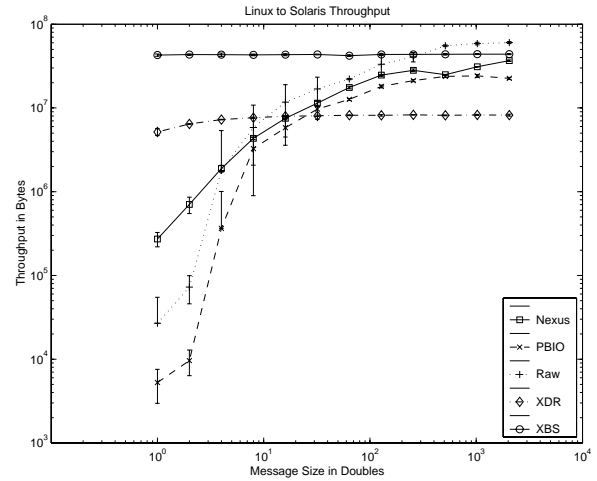
XBS performs much better than a number of other serializers when used in situations where its necessary to send large, dynamic data structures. In such situations, message-based serializers necessarily decompose the data into a set of smaller, fixed messages, which incurs higher overhead costs.

8 References

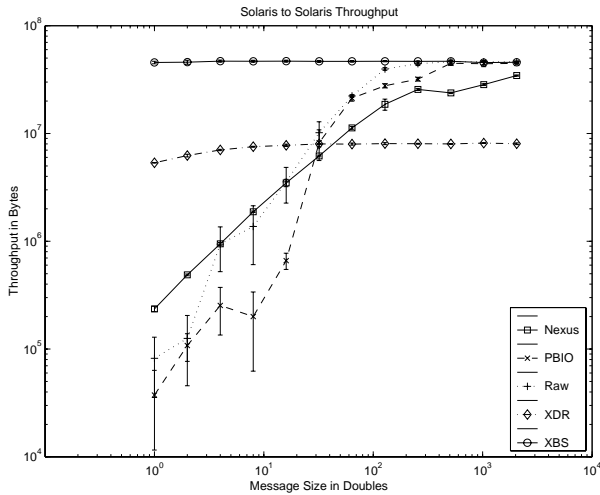
- [1] Bustamante, Fabian E., Eisenhauer, Greg, Schwan, Karsten, and Patrick Widener. "Efficient wire formats for high performance computing". In *Proc. of Supercomputing 2000 (SC 2000)*, Dallas, TX, November 2000.
- [2] Macklem, Richard. "Lessons Learned from Tuning the 4.3BSD Reno Implementation of the NFS Protocol". *Proceedings of the Winter USENIX Conference*, pp. 53-64, USENIX, Dallas, TX (January 1991).
- [3] Kiczales, Gregor, et. al. "An overview of AspectJ". In *Proceedings of the European Conference on Object-Oriented Programming*. Budapest, Hungary, 18--22 June 2001.
- [4] Foster, Ian, Kesselman, Carl, and Tuecke, Steven. "The Nexus Approach to Integrating Multithreading and Communication". *Journal of Parallel and Distributed Computing*. 37(1), 1996.
- [5] Object Management Group, *Common Object Request Broker Architecture: Core Specification*. http://www.omg.org/technology/documents/formal/corba_iiop.htm.
- [6] Srinivasan, Raj. *XDR: External Data Representation Standard*. <http://www.ietf.org/rfc/rfc1832.txt>.
- [7] Srinivasan, Raj. *RPC: Remote Procedure Call Protocol Specification Version 2*. <http://www.ietf.org/rfc/rfc1831.txt>.
- [8] Westhead, Martin and Bull, Mark. "Representing Scientific Data on the Grid with BinX". EPCC, January 2003. <http://www.epcc.ed.ac.uk/~gridserve/WP5/Binx/sci-data-with-binx.pdf>.



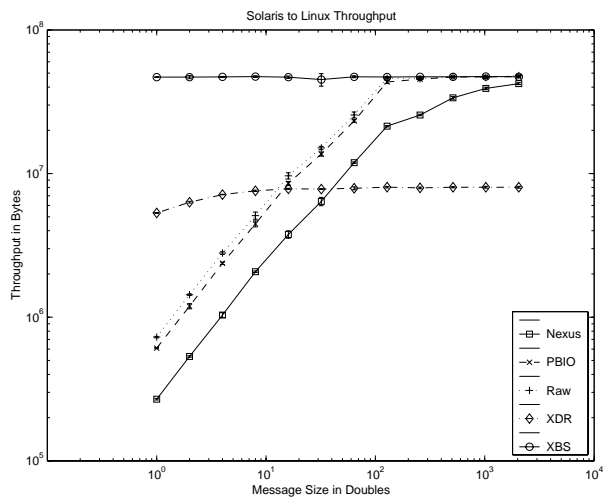
(a)



(c)



(b)



(d)

Figure 3. These graphs show the relative performance of XBS compared to other common binary serializers. In (a), the transfer was between two Linux machines. In (b), the transfer was between two Solaris machines. We note that small reads on Solaris seems to have poor performance and high variance. In (c), the transfer was from a Linux machine to a Solaris machine. Interestingly, the hand-coded raw serializer performed significantly better than XBS for larger messages. In (d), the transfer was from a Solaris machine to a Linux machine. Note that small writes on Solaris do not have the same problems as small reads.

The Linux machines were 2.8GHz Pentium Xeons. The Solaris machines were Sun Fire 280R equipped with 1.2GHz SPARC III CPUs. All machines were interconnected via switched Gigabit Ethernet.