Jdib: Java Applications Interface to Unshackle the Communication Capabilities of InfiniBand Networks

Wan Huang[1,2], Hongwei Zhang[1,2], Jin He[1] , Jizhong Han[1], Lisheng Zhang[1]

[1.] Institute of Computing Technology, Chinese Academy of Sciences

[2.] Graduate University, Chinese Academy of Sciences

{huangwan, zhanghongwei,hejin, hjz, zhang}@ict.ac.cn

Abstract

Using existing TCP/IP emulation layers over InfiniBand networks in Java applications presented rather poor communication performance and heavy overhead in host CPU. In this paper, we propose Jdib (Java Direct InfiniBand), a Java application acceleration technique to directly exploit RDMA (Remote-Direct-Memory-Access) capability of InfiniBand network hardware. The preliminary test results show that with Jdib API, micro-benchmark written by Java can achieve latency performance (~9us) comparable with that of micro-benchmarks written by C language (~8.5us) in 20Gbps InfiniBand fabrics.

Keywords

InfiniBand, Verbs API, JNI, RDMA

1 INTRODUCTION

For its portability, secure and easy-to-learn properties, Java has become the most popular programming language in the past 10 years. Nowadays, Java begins to play a role that C and C++ have been done successfully in last thirty years, especially in critical enterprise computing applications. To a great extent, Java is changing into a system-level programming language instead of the Internet specific programming language that it was designed to be.

With a byte-code executable, how to improve the execution efficiency and performance of Java applications' performance has been an active topic for a long time. Much efforts have been committed there. One of the most successful approaches is JIT (Just-In-Time technique). By caching the native code snippets of frequently used byte-code, JIT can help remarkably cut-down the numbers of the byte-code to native code translation and improve the performance of byte-code executions.[28][29]reported excellent (nearly the same as C/C++) Java application performances with JIT technique.

It is well known that the overall performance of applications is determined not only by the "pure" instruction execution efficiency but also by the I/O. For Java, I/O performance was not one of the primary design efforts obviously from the beginning because of its specification for security and garbage-collection memory management mechanism. So in a long time, the progress of Java I/O performance lagged far behind that of Java byte-code instruction execution optimizations. When Java is becoming a language much like C/C++, people have to confront the I/O performance problem in Java.

In recent years, more and more Java application servers have been deployed within data centers with InfiniBand Architecture fabrics (IBA) [1]. With industry efforts to offer low latency, high throughput and low CPU overhead characteristics inter-connection networks, the InfiniBand networks present two outstanding features: the Remote Direct Memory Access (RDMA) capability and dedicated protocol offload engines. Moreover, according to the InfiniBand industry community, OpenFabrics Alliance, InfiniBand can cut-down dramatically the system management costs for intelligent system management facilities embedded in the InfiniBand switches and host interface cards.

However, currently no direct support provided in Java to fully exploit the communication capability of InfiniBand networks. According to our recent research works [30], we found in Giga-bit Ethernet environment, the bandwidth and latency performances of Java micro-benchmark is nearly the same as that of C micro-benchmark, though with a bit higher CPU utilization. However because the only viable approach in Java to use InfiniBand is the TCP/IP emulation, our tests results with TCP/IP emulation layers over InfiniBand in Java and C micro-benchmarks show rather big gaps no matter bandwidth/latency or CPU utilization[4]. It suggests it is necessary and important to design a new Java API and supporting stuffs to make full use of InfiniBand in Java network applications.

In this paper, we propose Jdib (Java Direct InfiniBand), a Java application interface to directly exploit RDMA (Remote-Direct-Memory-Access) capability of InfiniBand network hardware. The preliminary test results show that with Jdib API, micro-benchmark written by Java can achieve latency performance (~9us) comparable with that of micro-benchmarks written by C language (~8.5us) in 20Gbps InfiniBand fabrics.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the details of design and implementation of Jdib. Section 4 gives the preliminary experimental results. Finally, Section 5 discusses open issues and future work, summarizes our findings and concludes the paper.

0-7695-2943-7/07 $25.00 © 2007 IEEE
DOI 10.1109/NPC.2007.111

2 BACKGROUND AND RELATED WORK

In this section, we first present an overview of InfiniBand. Then we discuss the related work.

2.1 InfiniBand Architecture

InfiniBand Architecture (IBA) [1] is an interconnect technology primarily used in high performance computing. The IBA specification defines a connection between processor nodes and I/O nodes such as storage devices. It is independent of hardware and software environments. IBA has many benefits. It affords high bandwidth, reliability and scalability features for I/O. It has extremely low latency and low CPU overhead. With IBA, applications directly access the IBA network hardware, without the interaction of operating systems (OS), which makes message-passing operation more efficient.

Two types of channel adapters are defined in IBA: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCA offers an interface through which the consumer can use the functions specified by IBA verbs. TCA is a programmable DMA engine that initiates DMA operations locally and remotely.

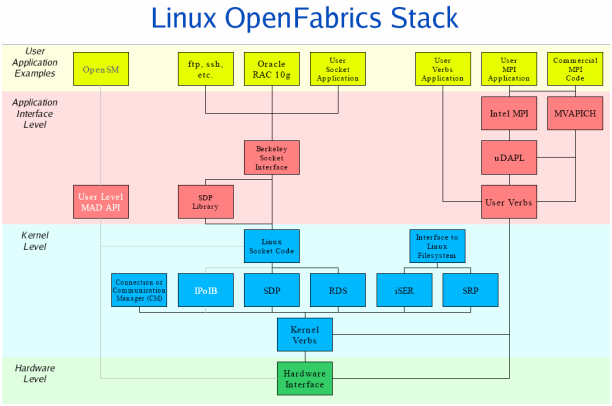

Figure 1: Linux OpenFabrics Stack (from [27])

Two useful communications are provided by IBA: channel semantics and memory semantics. The former, also called Send/Receive, is a classic communication style that one pushes the data and the other receives the data. The memory semantics, also known as RDMA, provides a way for an initiating node to directly read or write the virtual address space of a remote node. When the initiating node sends a read or write request, data are retrieved from or placed to the remote memory without the remote applications or OS involved. This is especially useful in massively parallel computing clusters.

2.2 Related Work

Javia [8, 9] in Cornell University is one of the earliest research projects about Java I/O performance. In Javia, a special Java-level buffer abstraction is provided. It allows Java applications to allocate regions of memory outside the Java heap (under the control of Java garbage-collection mechanism) and to use them directly and safely as Java arrays. Although targets both of disk and network I/O performance optimization, the major highlights of Javia is its Java interface to the VIA (Virtual-Interface-Architecture) network interfaces. As an fruit of joint industry research efforts from Intel, Microsoft and Compaq (now HP), VIA enabled applications to directly access network interface hardware with necessary security assurance and by-pass the kernel from the message exchange critical path so as to decrease host CPU utilization dramatically. The results of micro-benchmarks with Javia showed that Java applications could achieve almost the same of peak network communication performance as those of C applications.

Jaguar[5, 6],developed by Matt Welsh as a PhD candidate in UCBerkeley, is another important early research project in Java I/O performance optimization. Jaguar is much like Javia that its low-level interconnection is also a VIA network by UC Berkeley [7]. With JIT technology, Jaguar extended Java runtime to enable Java applications to directly access operating system and hardware resources instead of using JNI interface that was rather low-efficiency. JaguarVIA [6] is a Java application interface to the Berkeley VIA based on Jaguar. Performance tests showed that Java application achieved nearly identical performance of that of C. However, the Jaguar approach to extend JIT compiler raised a security concern, that's generated-code by Jaguar actually failed to preserve the type-safety properties of Java byte-code.

The work of Jaguar influenced the engineers of Sun Microsystems deeply after a wonderful presentation by Matt Welsh there. Java New I/O [10] (Java NIO) of JDK 1.4.2 in 2003, is the first efforts of Sun Microsystems to try to systematically improve the I/O performance of Java. Java NIO provided many more one-to-one mappings between Java API and host native API so as to decrease the semantic gaps and speedup the I/O executions. One of most outstanding features by Java NIO is the direct buffer. It's much alike a memory region allocated in C/C++ applications with malloc c function or new method and is out of the scope of Java garbage collection mechanism. No extra data copy operations are needed while sharing data between native operating systems and Java applications with the direct buffer. Now a new extension to NIO, naming NIO2 [11], is being developed under JSR 203.

Taboada [4] examined Java socket performance optimizations methodologies and proposed the Java Fast Socket (JFS) on their SCI (Scalable-Coherent-Interface) [3] testbed. The performance tests results demonstrated that it's possible that Java applications based on JFS can achieve performance closed to those of C application over standard SCI socket interface.

Other related research work of Java communication performance optimization included many efforts in the Java Remote Method Invocation (RMI [12]). RMI is a common communication facility for Java applications. Manta [13, 14] provided an optimized RMI interface with a Java to native code complier. JavaParty [15] proposed an optimized strategy of Java object serialization and an improved RMI inter-

face (KaRMI [16, 17]). Manta utilized native code for Java object serialization and de-serialization on Myrinet [31], and left out the type check, so it demolished the Java security. KaRMI got several data copy operations in critical data transportation path and resulted in poor performance. Java MPI [32] was introduced to replace RMI to provide high performance cluster communication mechanism in parallel computing environments. As a pure Java message-passing framework by University of Maryland, MPJava [20] harnessed the high-performance communication capabilities of Java NIO and delivered performance competitive with native MPI codes.

3 DESIGN AND IMPLEMENTATION

Java NIO provides us one scheme to access the network directly. Based on that, we can improve Java's communication performance by using specific protocol.

As we discussed above, the high overhead of IPoIB makes it difficult to fully exploit the performance of InfiniBand. Therefore, we focus on investigating how to utilize RDMA to improve the performance of Java communication.

3.1 JNI

Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the Java virtual machine into native applications. The primary goal is binary compatibility of native method libraries across all Java virtual machine implementations on a given platform.

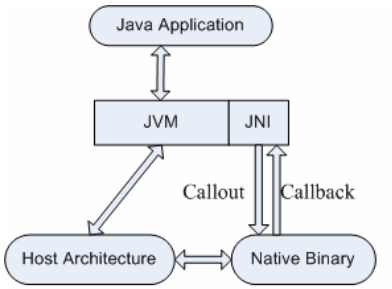

Figure 2: JNI Interface [33]

The most important benefit of the JNI is that it imposes no restrictions on the implementation of the underlying Java VM. Therefore, Java VM vendors can add support for the JNI without affecting other parts of the VM. Programmers can write one version of a native application or library and expect it to work with all Java VMs supporting the JNI.

The JNI framework allows a native method to utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code.

The JNI is used to write native methods to handle situations when an application cannot be written entirely in the Java programming language. For example when the standard Java class library does not support the platform-specific features or program library. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications. Many of the standard library classes depend on the JNI to provide functionality to the developer and the user, e.g. I/O file reading and sound capabilities. Performance- and platform-sensitive API implementations in the standard library allows all Java applications to access this functionality in a safe and platform-independent manner.

Based on the current IB verbs API implemented in C, we encapsulated it with JNI to provide a new version of IB verbs API. Java applications can perform the RDMA operations through this new interface to improve the performance of Java communication.

3.2 Implementation

IBA describes the service interface between a host channel adapter and the operating system by a set of semantics called Verbs. Verbs describe operations that take place between a host channel adapter and its operating system based on a particular queuing model for submitting work requests to the channel adapter and returning completion status. They also describe the parameters necessary for configuring and managing the channel adapter, allocating (creating and destroying) queue pairs, configuring QP operation, posting work requests to the QP, getting completion status from the completion queue.

The foundation of IBA operation is the ability of a consumer to queue up a set of instructions that the hardware executes. This facility is referred to as a work queue. Work queues are always created in pairs, called a Queue Pair (QP), one for send operations and one for receive operations. In general, the send work queue holds instructions that cause data to be transferred between the consumer's memory and another consumer's memory, and the receive work queue holds instructions about where to place data that is received from another consumer.

The encapsulating procedure can be described as follows: First, we constructed a java class named Device whose methods perform the operations needed by RDMA, described in the following table:

Table 2. Methods of class Device

Method	Description
Find IB device	
getDeviceList()	Get the device list
Initiation Operations	
openDevice()	Initialize device for use
queryDevice()	Get device properties
allocPD()	Allocate a protection domain
regMR()	Register a memory region
createCQ()	Create a completion queue
createQP()	Create a queue pair
modifyQPtoINIT()	Modify a queue pair to INIT state
initWR()	Init a work request
Create connection	
queryPort()	Get port properties
exchDest()	Exchange descriptions between server and client
modifyQPtoRTR()	Modify a queue pair to Ready-to-Receive state
modifyQPtoRTS()	Modify a queue pair to Ready-to-Send state
Data Transport	
postSend()	Post a list of work requests to a receive queue
pollCQ()	Poll a CQ for completions

This class performs as a bridge between the java applications and the C IB verbs library. Then the java applications can create an instance of the class Device and invoke its methods to perform the whole flow of RDMA operation. In addition, all the methods of class Device is implemented by native C code through JNI.

3.2 Design Decisions

We abstract the verbs API to a higher level to provide an interface much easier to use. The application programmers need not to maintain complex data structures representing the devices and connection contexts. They just need to specify the basic parameters for RDMA, such as IB port, bytes to transport, size of RDMA buffers and so on.

4 PERFORMANCE EVALUATION

4.1 Experimental Testbed

The experiments environment we used to conduct the performance testing is an 8-node cluster. Each node is a DELL SC430 pc server. The detailed hardware configuration of the DELL Power-Edge SC430 is as followings:

- CPU: Intel Pentium-4 2.8GHz, Hyper-Thread enable
- DRAM: DDR II 400, 1GB
- Chipset: Intel E7230 and ICH7R
- On-board Giga-bit Ethernet Controller: Broadcom NetXtreme Tg3
- InfiniBand HCA: Mellanox MT25204, Single Port, 20Gbps

The software installed in each node contains:

- RedHat AS 4.0 with Linux kernel 2.6.9-42.ELsmp
- InfiniBand drivers and library: OFED-1.2
- Sun JDK: 1.6-b-105

The shared facilities of the cluster include:

- InfiniBand Switch: MT47396, 24 port, 20Gbps/port
- Ethernet Switch: HUAWEI Quidway S1224, 24 port, 1Gbps/port
- NFS Server (shared home directory): DELL SC1420 with High-Point S-ATA RAID

4.2 Jdib Performance

Figure3 and Figure4 compare the RDMA read performance obtained by using C and Java verbs API (Jdib).

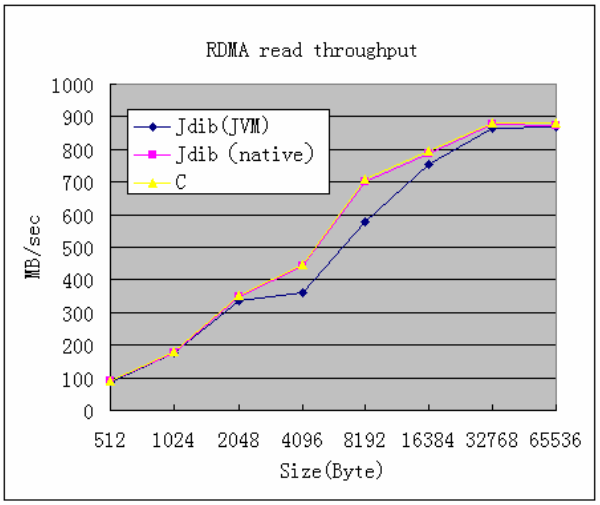

Figure 3: RDMA read throughput

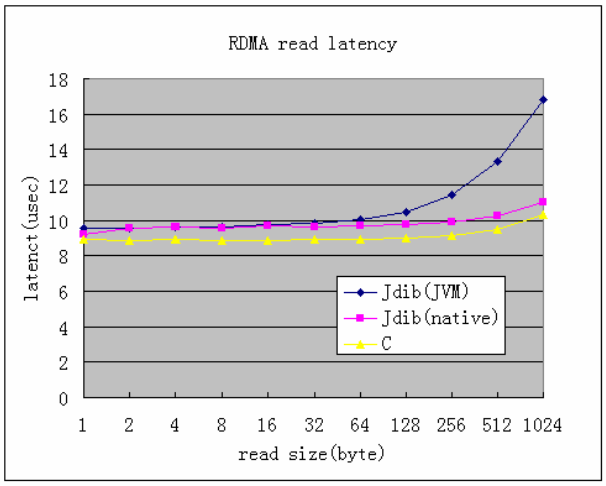

Figure 4: RDMA read latency

The difference between Jdib (native buffer) and Jdib (JVM buffer) is that the former stored the data in the buffer allocated in native code, while the latter additionally copied them to the JVM buffer through the JNI. The extra copy through JNI cost some time. The JNI's strength lies in decoupling native codes from a specific JVM implementation by providing relatively opaque access to JVM internals, data, and services. The cost of this property is efficiency, namely large runtime overheads during callouts to native functions, and even larger ones during callbacks to access Java code and data.

5 CONCLUSION AND FUTURE WORK

In this paper, we explored the viable approach to make use of the low-level InfiniBand user verbs API in Java applications and proposed the Jdib API based on standard Java JNI interface. From the test results, it's concluded that it's possible to exploit the potential capability of InfiniBand network in Java applications with the help of Jdib. The preliminary performance test revealed that latency performance difference between Java runtime based on Jdib and C language is trivial.

However, at the same time, we observed the Java JNI is rather poor of parameters passing between Java runtime and native world. We deemed there are two possible approaches in future to attack this problem. The first is quite straightforward: based on open-source Java, to make the low-level emulation layer architecture be more adaptable with Java runtime semantics and behaviors, and to get a better performance than that of now. The Second is to use direct-buffer mechanism provided by Java NIO in the Java Verbs API. The direct buffer is allocated outside of the normal garbage-collected heap, even outside the JVM process space. Thus, JVM can perform native I/O operations directly upon direct byte buffers, avoiding the copy of buffer's content using an intermediate buffer whenever an invocation of one of the underlying operating system's native I/O operations is done. We expect the latency gap between Java and C to be reduced after using this direct-buffer. Currently we have commenced work on designing a Jdib version based on Java NIO.

ACKNOWLEDGMENTS

This study was supported by funds from the China National 973 program (2004CB318202). The project team thanks Professor Chengde Han for his support. The work of Zhiying Jiang, Nan Wang and Yonghao Zhou to help setup the test-bed is highly appreciated.

REFERENCES

[1] A. H. Pajjuri A., "A Performance Analysis of Java and C," 2001.

[2] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The Virtual Interface Architecture," Micro, IEEE, vol. 18, pp. 66-76, 1998.

[3] K. Alnaes, E. H. Kristiansen, D. B. Gustavson, and D. V. James, "Scalable Coherent Interface," 1990, pp. 446-453.

[4] G. L. Taboada, J. Tourino, and R. Doallo, "Designing Efficient Java Communications on Clusters," 2005, pp. 182a-182a.

[5] M. Welsh, "Safe and Efficient Hardware Specialization of Java Applications," 2000.

[6] D. C. M. Welsh, "Jaguar: Enabling Efficient Communication and I/O from Java," in Concurrency: Practice and Experience, 1999.

[7] B. Philip, G. Andrew, and C. David, "An implementation and analysis of the virtual interface architecture," in Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM) San Jose, CA: IEEE Computer Society, 1998.

[8] C.-C. Chang, "Safe and Efficient Cluster Communication with Explicit Memory Management," 2000.

[9] C.-C. Chang and T. v. Eicken, "Javia: A Java interface to the virtual interface architecture," 2000.

[10] Sun Microsystems, "Java New I/O," p. http://java.sun.com/j2se/1.4.2/docs/guide/nio/, 2002.

[11] Sun Microsystems, "More New I/O APIs for the JavaTM Platform ("NIO.2") ", 2006.

[12] Sun Microsystems, "Java Remote Method Invocation Specification," 1997.

[13] R. v. N. J. Maassen, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for parallel programming," Programming Languages and Systems, vol. 23(6):747-775, 2001, 2001.

[14] M. Jason, N. Rob van, V. Ronald, E. B. Henri, and P. Aske, "An efficient implementation of Java's remote method invocation," in Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming Atlanta, Georgia, United States: ACM Press, 1999.

[15] B. H. M Philippsen, C Nester, "More efficient serialization and RMI for Java," in Concurrency Practice and Experience Chichester, West Sussix: John Wiley & Sons, Ltd, 2000.

[16] U. Karlsruhe, "KaRMI: Efficient RMI for Java."

[17] N. Christian, P. Michael, and H. Bernhard, "A more efficient RMI for Java," in Proceedings of the ACM 1999 conference on Java Grande San Francisco, California, United States: ACM Press, 1999.

[18] N.-B. C. Laboratory, "MPI over InfiniBand Project."

[19] I. T. Association, "InfiniBand Trade Association."

[20] B. Pugh and J. Spacco, "MPJava: High-Performance Message Passing in Java using Java.nio," "in Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC'03), College Station, TX, 2004.

[21] "The Public Netperf Homepage," http://www.netperf.org/netperf/.

[22] wikipedia.org, "Sockets Direct Protocol."

[23] P. B. S. B. H.-W. J. D. K. Panda, "Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (SDP) over InfiniBand," in Workshop on Communication Architecture for Clusters (CAC); in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS) Rhodes Island, Greece, April, 2006., 2006.

[24] InfiniBand Trade Association, InfiniBandTM Architecture Specification Volume 1, Release 1.2,2004.

[25] Sheng Liang , The Java Native Interface Programmer's Guide and Specification, 1999.

[26] Mellanox Technologies, Inc. InfiniHost III Programmer's Reference Manual, 2005.

[27] Johann George, QLogic, "A Tour of The OpenFabrics Stack", June 2006, presentation of 2006 OpenFabrics Workshop.

[28] O. Hirotaka, S. Kouya, M. Satoshi, M. Fuyuhiko, S. Yukihiko, and K. Yasunori, "OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java," in Proceedings of the 14th European Conference on Object-Oriented Programming: Springer-Verlag, 2000.

[29] T. O. T. Suganuma, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, "Overview of the IBM Java Just-in-Time Compiler ": research.ibm.com, 2000.

[30] Hongwei Zhang, "A Performance Study of Java Communication Stacks over InfiniBand and Giga-bit Ethernet", technical report, 2007.

[31] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and S. Wen-King, "Myrinet: a gigabit-per-second local area network," IEEE Micro, vol. 15, pp. 29-36, 1995.

[32] N.-B. C. Laboratory, "MPI over InfiniBand Project."

[33] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining java native calls at runtime. In VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments.