UNIVERSITETET I OSLO
Institutt for informatikk

Creating and evaluating a Distributed Virtual Machine using zero-copy RDMA operations over InfiniBand networks

Master thesis

Mathias Myrland

February 2012

Contents

1 Introduction **1**

1.1 Introduction to the Distributed Virtual Machine 3

1.2 On InfiniBand and the IBPP 4

1.3 Problem definition 5

1.4 Development methodology 6

1.5 The tools 6

2 Background **9**

2.1 A brief introduction to InfiniBand 9

2.2 Existing DSM systems 16

2.3 MPI and InfiniBand 18

2.4 OpenMP 19

2.5 Distributed runtime systems 20

2.6 Other Virtual Machines 21

3 The InfiniBand++ framework **23**

3.1 An introduction to the libibverbs interface and the rdma_cm API 24

3.2 Designing the IBPP framework 25

3.2.1 The IBManager 26

3.2.2 Memory management 28

3.3 An example program 29

3.4 Advanced usage and working with RDMA buffers 32
3.5 What remains to be done 34
3.6 IBPP performance evaluation 34
3.6.1 Memory registration performance 35
3.6.2 Send and Receive latency 37
3.6.3 RDMA Read 39
3.6.4 RDMA Write 42
3.6.5 RDMA Swap-Compare evaluation 44

4 The Distributed Virtual Machine 47
4.1 Architecture 47
4.1.1 The instruction set 48
4.1.2 The application 50
4.1.3 The memory architecture of the DVM runtime system 50
4.1.4 The runtime environment 50
4.1.5 The dispatcher 51
4.1.6 The page node 51
4.1.7 The computing node 53
4.1.8 The network topology file 54
4.2 DVM performance 55
4.2.1 Register-to-register performance 56
4.2.2 Push and pop performance 57
4.2.3 Page locking and swapping 59
4.2.4 Shared memory access 60

5 Discussion 65
5.1 The dispatcher 65
5.2 The runtime environment 67
5.3 Distributed assembly 68
5.4 The shared memory model 68
5.5 Improving the IBPP framework 69
5.6 Writing a compiler framework for the DVM 70

5.7 Application control 70
5.8 Local resource sharing 71
5.9 Threading 72
5.10 Going big - large scale applications and clusters 73
5.11 Some final thoughts 75

6 Conclusion **77**
6.1 IBPP contributions and future work 78
6.2 InfiniBand and RDMA as a platform for DSM systems 78
6.3 The future of the DVM and closing words 79

A The DVM instruction set **i**

List of Figures

2.1 The InfiniBand protocol stack 10
2.2 InfiniBand QP and Completion Channel communication . . . 13
2.3 RDMA mapping 14

3.1 Consumer interaction with the IBManager 26
3.2 Cost of allocating and registering RDMABuffer objects, overhead and de-registration cost 36
3.3 RTT versus message size 38
3.4 Read latency versus buffer size 40
3.5 Maximum achieved bandwidth versus buffer size 42
3.6 Write latency versus buffer size 43
3.7 Asynchronous write bandwidth versus buffer size 43
3.8 Atomic swap/compare access flow 44

4.1 Architecture of the DVM 48
4.2 DASS basic syntax forms 49
4.3 DASS operand syntax 49
4.4 Memory storage model for the DVM 52
4.5 Page swapping algorithm 53
4.6 Push and pop comparison for 1MB of data 58
4.7 Transfering 10MB using DVM memory operations and C memcpy 62

5.1 The Dragonfly topology 75

List of Tables

2.1 InfiniBand signaling and data rates 11
2.2 Supported operations for various transport modes 15

3.1 Achieved bandwidth in Mbit/s through message passing 39
3.2 Achieved bandwidth in MB/s through blocking RDMA reads . 39
3.3 IBPP read latencies for asynchronous and synchronous operations 41

4.1 DVM and C++ speed comparison over 1000000 iterations . . 57
4.2 Cost for acquiring ownership of a page for the three states under race conditions 60

A.1 Register instructions ii
A.2 Memory instructions ii
A.3 Bitwise instructions iii
A.4 Arithmetic instructions iii
A.5 Control instructions iv
A.6 Threading and Synchronization instructions v

Acknowledgements

This thesis is the product of a lot of support, encouragement and patience from many people. First and foremost I would like to express my deepest gratitude to my supervisors, Sven-Arne Reinemo and Tor Skeie for their faith in me and belief in the project, and for making this idea come to life.

I also wish to thank my family and friends for their continued support and encouragement throughout this period. I would not have been able to do this without you.

Finally, thank you to all my fellow students at the ND-lab, past and present, for being the best colleagues one could possibly wish to have, despite being a great source of procrastination.

Chapter 1

Introduction

Computers have always been used as tools to process large amounts of data. Ever since the very beginning of the computer industry, scientists have used computers to process large data sets and results from sensor observations. The banking industry use them to conduct hundreds of thousands of transactions per minute, and businesses all over the world rely on them to handle their databases. Needless to say, the requirements these applications place on systems, both for storage, processing power and available system memory, is growing rapidly.

In the beginning of the industry, computers relied on a single CPU and limited system memory. As processor architecture developed, having multiple processing units on a single system became common. Applications could benefit from concurrently executing multiple threads of code simultaneously to speed up their calculations, and the size of available physical memory also grew rapidly over the years. But the demand continued to grow beyond what a single system is capable of, and developers started using multiple

machines connected with high speed networks, or clusters, to facilitate their applications.

The programming task of distributing the computations and migrating data between the different nodes in the cluster, can be overwhelming. There are a lot of additional factors to account for when programming a distributed application for a cluster, than what you have to consider when writing code for a single machine. The programmer needs to explicitly implement communication between the nodes, to synchronize execution and share data. Processing tasks need to be assigned on a per node basis, and a lot of care has to be taken to avoid race conditions and other problems related to network delay and errors. Especially for mathematicians, physicists or scientists from other fields that require large computational programs to be developed, creating code for distributed systems is non-trivial and requires a lot of effort.

A few libraries and standards has emerged over the years, to ease the work load of implementing clustered applications. One of the first libraries which really took this into consideration was the Parallel Virtual machine (PVM)[18]. It provided functions for passing messages between nodes, spawning jobs on remote hosts and synchronizing data flow. A few years later, the MPI standard was specified in 1994[52]. It provided a standard for message passing between nodes in a cluster, and quickly became the accepted standard for distributed programming.

Other approaches has been taken to ease the development process for distributed systems. One example is the Emerald[20] programming language, which is designed to have the distribution of objects built into the runtime of the application. This makes object migration easier to handle, since it is built into the language specification.

As we will discuss later, the idea of Distributed Virtual Machines is not new, and several attempts has been made to implement one. The driving force behind this area of research is to hide the distribution under a Single System Image (SSI), where the application has the perception that it is running on a single system rather than a cluster of nodes.

The idea for this project, and eventually thesis originated during a lecture on the runtime systems of programming languages. In the beginning, the idea was to create a distributed language runtime, not at all dissimilar from that of Emerald. As it grew and expanded, the idea of using the Remote Direct Memory Access(RDMA) capabilities of our InfiniBand(IB)[31] research cluster was proposed by Sven-Arne, and the birth of InfiniBand++(IBPP) was a fact.

I quickly leaned towards using a custom virtual machine, since it both stimulated my interests as a programmer and gave more control over the implementation than using an existing system would. In hindsight, using an existing JavaVM such as Kaffe[10] would probably have been easier and more efficient implementation-wise, but this is not the direction the project took.

1.1 Introduction to the Distributed Virtual Machine

The goal of the Distributed Virtual Machine(DVM) project is to make programming for and utilization of the shared memory and processing resources of large interconnected clusters easier and more accessible than the established distributed programming schemes, such as MPI or OpenMP. The idea

behind the project is to create a bytecode and assembly language that can utilize the Distributed Virtual Shared Memory[46] space of an InfiniBand cluster.

By having a bytecode that can be run on several physical machines without any explicit consideration of the distributed properties, we allow for standard single machine multi threaded programming paradigms to be used for creating applications that utilize the resources only a large cluster can provide. Additionally, any cluster capable of compiling the virtual machine should be able to execute the programs written for the DVM.

1.2 On InfiniBand and the IBPP

As previously mentioned, I was introduced to IB when I started my work on this thesis. A lot of time and effort has gone into learning and getting familiar with the verbs interface, and the inns and outs of the IB model and software stack in general. As I learned and developed my understanding, I expanded my code base to become a general-purpose library, resulting in the IBPP framework.

The RDMA programming model provided by the InfiniBand architecture fits the requirements of an application such as the DVM nicely, both in terms of available bandwidth, scalability and latency. But most importantly, the one sided zero-copy functionality provided by using the RDMA operations prove to be very fitting for this task.

1.3 Problem definition

The goal of the project has from the beginning been to implement a custom Virtual Machine capable of running on a cluster of computers, providing a SSI environment for its guest applications. To achieve this, there are key problems that need to be addressed and solved. We need to design and create a communication framework capable of RDMA operations in order to implement the distributed memory for the DVM. The VM itself has to be designed and implemented to utilize the shared memory, and the shared memory system must be created. Additionally, an assembly language with support for the specific properties of the VM has to be devised.

For the DVM, the core problems that we need to solve are the following:

- Design a runtime system and assembly language suitable for the distributed VM.
- Implement an efficient interpreter to execute the bytecode of the VM.
- Design a shared memory system that can be accessed from every node executing bytecode, utilizing RDMA operations.

The last requirement, taking advantage of RDMA capabilities using the IB platform, introduces the IBPP subproject. This is a library intended to make working with IB and RDMA easier and more streamlined than using the vendor-provided library functions. For the IBPP, the development goals was to:

- Design and implement a user-friendly set of classes and functions, that set up, manage and enables IB communication.

- Make the library handle all the IB-related events and processing in the background, hidden from the user.
- Hide as much as possible of the underlying functionality by abstraction.

1.4 Development methodology

The development process during this project has been implementation-centric, and the two parts, IBPP and the DVM was developed in parallel. Features of the IBPP library has been implemented as they have become necessary for the DVM to operate. This proved to be an efficient way of testing the library while it was in development.

An alternate approach would have been to create a simulation of the DVM, using expected values for operations such as page swapping, memory access times and bytecode efficiency. While this might have sufficed for a proof of concept, or to evaluate various approaches, I feel that the direct implementation route was the better of the two options. With the DVM being fairly modular, testing new solutions for and modifications to the underlying Distributed Shared Memory system is possible with little impact on the VM core.

1.5 The tools

All the code developed for this thesis is written in C++, and compiled with *gcc 4.1.2*. The code uses many of the boost[9] 1.42.0 libraries. Additionally,

our test cluster is running the OFED[6] software stack version 1.5.1, which is a collection of drivers, libraries and utilities for IB clusters.

Chapter 2

Background

The programming done in this thesis spans several topics, from InfiniBand specific implementation of networking, to virtualization and distributed memory theory. This chapter aims to give a rudimentary background of the existing research within the topics discussed further on in the thesis.

2.1 A brief introduction to InfiniBand

The InfiniBand(IB) network technology is a relatively new interconnection technology. It builds upon the Virtual Interface Architecture[33] communication model, providing zero copy remote direct memory access without going through the host kernel to access memory on remote machines. The VIA model was presented as the product of a joint effort between Microsoft, Compaq and Intel in 1997. Its purpose was to create a low latency high per-

formance network technology primarily targeting high performance systems such as clusters, servers and supercomputers.

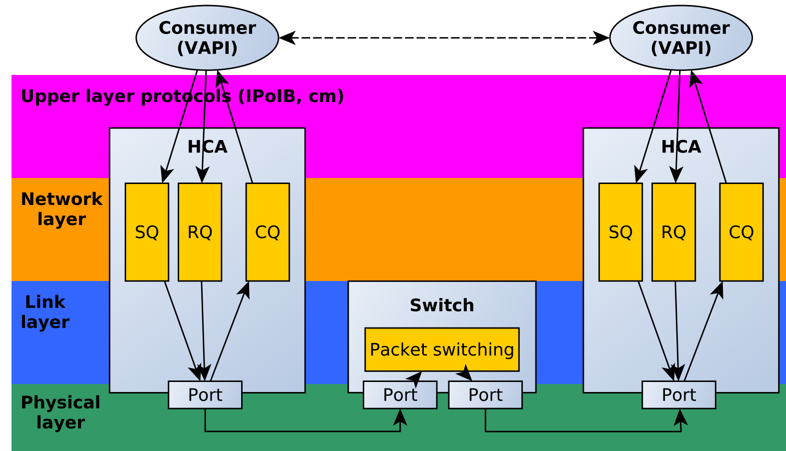

Figure 2.1: The InfiniBand protocol stack

The driving force behind the VI architecture was the need for zero-copy operations, which allow user mode applications to interface directly with the network adapter, without the need for going through the kernel for communication.

After its introduction in 2000, IB has been rapidly gaining popularity for use in HPC clusters and data centers, and IB enabled systems are now a regular entry on the HPC top 500 list [11]. In 2011, 42.6% of the systems on the list used IB interconnection.

Offering very low latency (as low as 1.07μs MPI latency) and very high bandwidth, InfiniBand is a widely accepted and rapidly growing interconnection solution for large data centers, clusters and other applications where network performance is of importance.

The IB fabric is the physical links and switches connecting the nodes of the

Data rate	Signalling rate 1x	Signalling rate 4x	Data rate 1x	Data rate 4x
SDR	2.5Gbit/s	10Gbit/s	2Gbit/s	8Gbit/s
DDR	5Gbit/s	20Gbit/s	4Gbit/s	16Gbit/s
QDR	10Gbit/s	40Gbit/s	8Gbit/s	32Gbit/s
FDR	14.0625Gbit/s	56.25Gbit/s	13.64Gbit/s	54.54Gbit/s
EDR	25.78125Gbit/s	103.125Gbit/s	25Gbit/s	100Gbit/s

Table 2.1: InfiniBand signaling and data rates

network together. Being a switched fabric, it offers the high performance necessary for modern interconnection networks.

There are variable data rate fabrics available, Single Data Rate (SDR), Double Data Rate (DDR), Quad Data Rate (QDR), Fourteen Data Rate (FDR) and the most recent Enhanced Data Rate (EDR). Table 2.1 show the maximum bitrate and actual data rate for the various connection rates. The data rate takes the communication overhead into account. For SDR, DDR and QDR this is 8/10, i.e every 10 bits of communication carries 8 bits of actual information, and for FDR and EDR it is 64/66. Multiple links can be aggregated into a single connection, offering either 1x, 4x or 12x bandwidth.

When working with the IB architecture, we need to come to terms with a key set of concepts and ideas. These are frequently used and referenced throughout this thesis, and a full listing can be found in the IB specification[31]. Here is an introduction to the most common of those:

Consumers are the applications using, or "consuming", the functionality of the verbs API and the fabric.

The Host Channel Adapter (HCA) is the PCI Express card responsible for

communicating with the fabric. It also maintains a direct mapping table from virtual to physical addresses to be used with RDMA operations.

The Communication Manager (CM) is a piece of software and/or hardware which facilitates setting up connections to destination ports on the network. Each port on an end node in the network is associated with its own CM, which takes care of maintaining QP connections. The QP creation and setup is done out-of-band using a low performance network to initialize the QP connection.

Operations over the IB fabric are performed by posting work requests to special queues, that in turn are processed by the HCA. Figure 2.1 illustrates this flow. Each connection over the IB fabric is represented by a pair of queues, called a *Queue Pair(QP)*. The send queue(SQ) is dedicated to send operations, and the receive queue (RQ) to receive operations. A third queue called the Completion Queue, is used to notify the consumer that a work request has been completed by the HCA.

Whenever we want to perform an IB-operation, we have to post *Work Requests (WRs)* to the appropriate queue. Work requests describe the details of the operation we wish to perform, such as Scatter-gather entries for the data, addresses, access keys and so on. Each work request is an instance of a struct, either *ibv_send_wr* or *ibv_receive_wr*. The content of these struct varies depending on the operation they are used to represent, and they are described in full in the RDMA aware programming user manual[42].

Scatter/Gather-entries tells the WRs where to store or find the data that is used/received. They can be lists of fragmented memory regions.

The Completion Queue (CQ) is filled up with Work Completion (WC) structs describing all completed WRs for the queues associated with that CQ. Work completions must be polled from the CQ.

Multiple CQs can post completion events on a completion channel. One event can consist of multiple work completions for the responsible CQ, and the notification must be requested after each processed event. Events on the completion channel must be acknowledged. Acknowledging completion events is a relatively expensive operation, but since the *ibv_ack_cq_events* call allows us to process more than one acknowledgement at the time, we can do it periodically to save time.

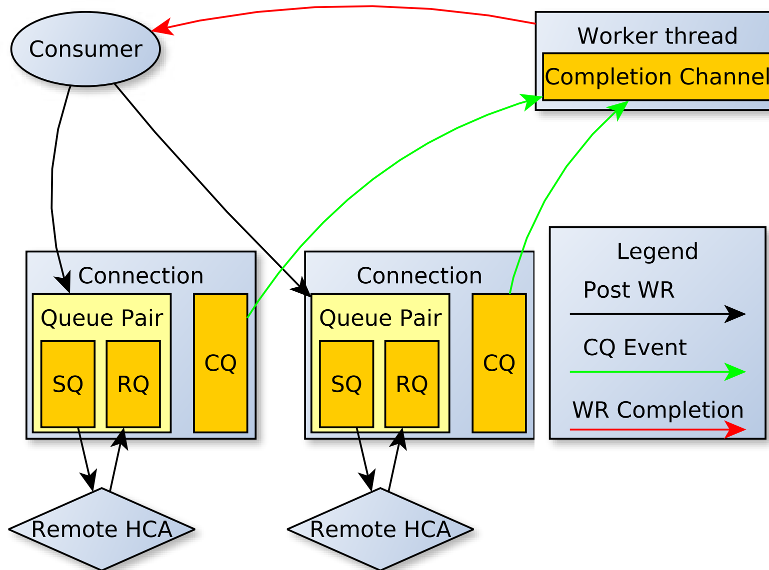

Figure 2.2: InfiniBand QP and Completion Channel communication

Figure 2.2 illustrates the flow of work requests in the system. First, the consumer posts a request, which is either a send or receive operation, to the corresponding queue in the HCA. The HCA then performs the action, either actively by sending the request over the fabric, or by waiting for a receive request to be consumed. When a request has been completed, the HCA posts

a completion entry in the completion channel associated with the queue pair, and the worker thread will be notified of the entry through a rdma channel event. The worker thread then in turn notifies the consumer by flagging the operation as completed, and invokes a callback in the completed operation.

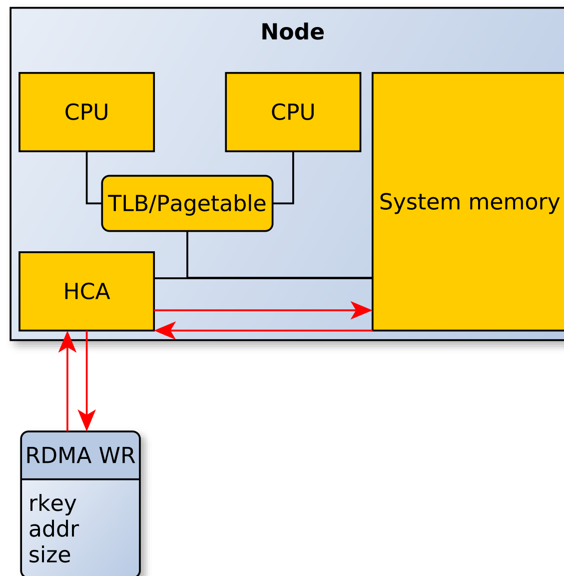

Figure 2.3: RDMA mapping

Remote Direct Memory Access (RDMA) allows you to access host memory directly through the HCA, without going through the host CPU and kernel. In order to utilize RDMA operations on a segment of memory, the memory region that is being accessed has to be mapped to the HCA, and a remote key needs to be sent to the node performing the operation. Figure 2.3 shows the path an RDMA request takes when the memory has been pinned and mapped in the HCA.

The buffers in memory that are pinned by the DMA and registered with the HCA are referred to as Memory Regions. All memory that is used for IB operations has to be registered and pinned in this fashion in order to ensure that they are present and to provide the virtual-to-physical address mapping

Operation	UD	UC	RD	RC
Send	x	x	x	x
Receive	x	x	x	x
RDMA Write		x	x	x
RDMA Read			x	x
Atomic operations			x	x

Table 2.2: Supported operations for various transport modes

required, since the DMA is oblivious to the virtual address space of the host machine. When a MR is created, it is initialized with a set of permissions for local write, remote write, remote read, atomic and bind. It also receives two keys to be used for accessing the memory, one is used by the local HCA to access the physical memory, and the remote key is distributed to and used by the remote HCAs performing RDMA operations on the memory region.

InfiniBand offers 4 different transport modes. Reliable Datagram, Unreliable Connection, Reliable Datagram and Reliable Connection. Not all operations are supported by every mode of connection. The compabilities can be seen in table 2.2. The Reliable Connection and Unreliable Datagram communication modes very closely resembles traditional TCP and UDP communication patterns. RC is the most popular connection mode for message passing, and is also the one used for the DVM developed in this thesis. It provides the same connection oriented and reliable performance as TCP. Unreliable and reliable datagrams allow a QP to communicate with multiple target QPs. The unreliable version does not guarantee ordering of packets, and the maximum message size is limited to the MTU of the adapter.

2.2 Existing DSM systems

A lot of research has been done in the area of Software Distributed Shared Memory(SDSM) systems. SDSMs build upon the shared memory system, proposed by Li and Hudak in [38]. They combine the memory of all the processing nodes on a cluster into a single, large virtual memory space, and allow programmers to access a global memory from any node in the system.

A SDSM system is in principle extensions of the hardware shared memory systems, which were developed to support multiple processor machines. Symmetric Multiprocessor (SMP) systems provide a shared memory solution, where all processors have access to the same main system memory. At the basic level, a SDSM system aims to replicate the properties of hardware based shared memory. They aim to provide global access to the memory of all nodes connected to the system, as transparently as possible. Transparency, in this context, means that the extra code imposed on the application utilizing the memory should be as small as possible. Ideally, there should be no additional code required to utilize the DSM. A SDSM system which is completely transparent to the user has what is called a Single System Image(SSI) property, which is the illusion of the entire memory existing in a single physical system.

In [30], Gharachorloo explains and discuss various problems related to developing software for and measuring the performance of various SDSM implementations. One of the main problems related to most SDSM systems is that they do not preserve the SSI hardware based shared memory systems benefits from. Most of them require explicit calls to access the shared memory, making it hard and expensive to convert code to utilize the memory. The initial development cost also goes up, since the code itself is harder to write and debug properly.

There are various approaches to memory consistency in DSM systems. Many of them are described in [13], which is an introductory tutorial to the various models. A sequentially consistent memory model guarantees that all memory access will happen in order, with global visibility. This means that the result of a write operation will have to be visible to all processes before the next memory operation is allowed. This imposes a lot of limitations on the programs, as verification of completion of writes will have to be sent before the next memory operation can be executed. There are various models with a more relaxed memory model than sequential consistency, and we will not describe every single one in detail here. The most common model is the Release Consistency (RC) in its various forms, the most common of which being Lazy Release Consistency[35], and its Home Based variations [54],[56],[22]. In a RC model, changes to the memory are propagated as diffs at latest at the first access after releasing the memory area, and at the earliest on the release itself.

IB and other VIA-based communication platforms such as iWARP[23] are well suited for DSM systems due to their zero-copy RDMA capabilities. Panda and Noronha investigates how RDMA operations can be utilized to improve the performance of DSM systems in [44]. Rangarajan and Iftode explores VIA used to implement DSM in [48]. ViSMI[45] is an IB-based DSM implementation. In [39], the possibility of exploiting kernel-space access to the IB verbs is investigated and used to implement a software DSM system in kernel mode.

The Distributed Virtual Machine described later in this thesis builds upon a hybrid implementation of a SDSM, resembling a strictly ordered home based lazy release model.

2.3 MPI and InfiniBand

The Message Passing Interface(MPI)[52] standard, and its expanded successor MPI2[51] has been the de-facto standard for programming massively parallel applications for the majority of this century. The specification provides a set of functionality well suited for sharing messages between processes over a cluster environment, and its relative ease of use compared to a more direct network approach has granted it the well-deserved popularity it has today.

A typical MPI program operates by executing a binary on multiple nodes, which communicate through the passing of messages. The various MPI implementations provide daemons and utilities that will take care of starting the execution on the desired set of nodes, often specified in a host file. After initialization, the nodes will determine what their task is based on processor ID (the ID of the current node), or by passing control messages. All data access is done by either message passing, or one sided put/get-operations, the latter being optimized as RDMA operations for RDMA capable architectures. Synchronization is done through the use of barriers and other synchronization mechanisms defined by the MPI specification.

There are currently two major implementations of the MPI2 standard, both supporting InfiniBand to a certain degree. OpenMPI[29] and MVAPICH2[4] both currently supports and utilize IB optimizations, either over IPoIB or directly through the verbs interface, it has low message latency and takes advantage of the hardware supported multicast, but they do not provide a direct programming interface to the verbs API.

MPI is well suited for developing distributed applications, and it is a fairly

accessible standard. Nevertheless, awareness of the distribution is required from the programmers point of view.

2.4 OpenMP

OpenMP[7] is a popular compiler framework for developing multithreaded applications. Originally developed for use on shared memory multiprocessor systems, it provides primitives to generate parallelized code using a single virtual memory space.

The basic idea behind OpenMP is that parallelizing certain parts of the code can be done by the compiler. This is achieved by providing a set of compiler directives that are used by the application developer to mark the code that is intended for parallelization. The OpenMP compiler then generates a multithreaded implementation of the source code, and passes it on to the machine code generating compiler. The programming model aims to be simpler than standard multi threaded programming paradigms, but the syntax is cumbersome and requires a lot of effort for non-professional programmers such as physicists and mathematicians to learn and use properly. It also has limitations regarding what can and cannot be parallelized. For instance, parallelizing code which has flow-dependency[41] is impossible when using OpenMP[1]. An example of flow dependent code can be seen in listing 2.1, where the calculation depends on steps taken previously in the iteration.

Listing 2.1: A flow dependent iteration

```
int arr[10];
for ( int i=1; i < 10; i++ ) {
  arr[i] = arr[i-1];
}
```

As the need for development tools for larger scale interconnected clusters arose, researchers began to extend OpenMP to generate code capable of running on a shared memory cluster. Omni/SCASH[49] is an example of this work. Omni/Infini[53] takes this work one level further, and builds an OpenMP framework on top of the ViSMI DSM system for InfiniBand. Additionally, Intel is working on of a clustered version of OpenMP called OpenMP*.

2.5 Distributed runtime systems

The final distributed programming scheme we will address here, is the distributed runtime approach. A distributed runtime system is a hybrid solution, where the compiler and/or the virtual machine of the language creates operations for object and thread migration. The memory in distributed runtime languages is usually shared on a per-object basis. The best known example of this is the previously mentioned Emerald[20] programming language. Its development started in 1984, and it pioneered this approach to distributed computing. It offers language supported mechanisms for distribution, but the distribution is exposed to the user, for instance by the exposure of nodes and location in the language.

Another promising implementation is JavaSplit[27], which modifies the program on a bytecode level to offer transparent thread and object migration. Another benefit of JavaSplit is that it is run inside a standard JavaVM, making it highly deployable on existing systems.

2.6 Other Virtual Machines

Virtual Machines is a heavily researched topic. They date back to the IBM 7044 in 1965, which is the first known practical implementation of a virtual machine environment. Virtual machines have a variety of applications, from system emulation to enabling platform independent code.

Generally, VMs are divided into two categories. The process VM, which runs as a process inside an operating system, providing a standard platform for bytecode. Process VMs are used when either cross platform compability, or an additional security layer to protect the OS from the code being run is required. The system virtual machine is another approach to virtualization, where the VM emulates an existing hardware platform, enabling guest operating systems to be running inside the virtual environment. This kind of VM is useful in cases where we want to run an operating system on a platform which is not directly supported by the OS.

The most famous and well known process virtual machine today is the JavaVM platform. It was introduced in the early 1990s[2], but in the beginning, it suffered from slow execution speed compared to native code, since the only VM implementation was interpreter based. With the introduction of the HotSpotVM[3], and increasing hardware and operating system support for virtualization, Java has emerged as a highly popular development platform.

The .net framework with its Common Language Runtime[21] is another example of a popular process VM, gaining a lot of recognition and use in the current industry. Featuring modern languages, and a rich set of libraries, it is a powerful virtual development platform.

Another level of virtualization is created with the Hypervisor, or Virtual Machine Monitor (VMM) technology. This software runs as a layer between the system virtual machine and the hardware, enabling several concurrent guest operating systems to be running simultaneously on a single computer. The Xen[16] hypervisor is an example of a commonly used VMM. Other virtualization software such as VMWware[12] and QEMU[17] aims to emulate a host computer, running a guest operating system inside their virtual environment. With increasing memory and CPU power of modern server architecture, hosting virtual servers provides an additional safety layer, as well as the possibility to swap OS-images without any downtime on the server.

The idea of a distributed virtual machine is far from new. A few distributed JavaVMs has been implemented. Java/DSM[55], JESSICA2[57] and the cJVM[14] all extend the JavaVM to run transparently on a cluster, distributing objects and having transparent thread migration. The term used to describe these environments is Single System Image(SSI), meaning that the distribution is transparent to the applications. None of these projects investigate the use of InfiniBand or other VIA capable networks, and are based on traditional ethernet technology.

There is also a commercial implementation of an InfiniBand-based DVM called vSMP, produced by ScaleMP[50]. While this is proprietary and closed software, it demonstrates that the idea is viable and has applicability outside of academic circles. They are the market leader in what they call Aggregated virtualization, combining multiple x86 systems into an apparently single large SMP x86 platform.

Chapter 3

The InfiniBand++ framework

The IB architecture has no standardized programming API. The specification[31] provides a set of verbs which has to be present, but the implementation is left to the vendors. The most widespread implementation today is developed by the Open Fabrics Alliance[5], the OpenFabrics Enterprise Distribution(OFED)[6] software stack.

This lack of a standardized API and framework for the IB architecture sparked an interest for developing a C++ library, intended to abstract working with the fabric to a slightly higher level than what was provided by the bare-bones verbs interface given by the OFED-stack. Over the course of this project, a significant part of the programming work has gone towards making and optimizing this library. While it is sacrificing some performance compared to direct verb calls, in the form of slightly increased latency for operations, it provides a much easier interface to the operations through the use of automatic memory management, worker threads and abstraction.

3.1 An introduction to the libibverbs interface and the rdma_cm API

Before discussing the design and implementation of the IBPP library, we need to examine both how the ibverbs-api is designed, and what steps we need to take in order to set up and use the HCA for communicating over the IB fabric.

The basic process of setting up a link between two end nodes is very similar to what you would do when creating a regular TCP connection. One party blocks while listening on a port, while the other connects to the destination port. Using verbs, however, requires careful creation and configuration of the various components required to establish communication. Luckily, using the rdma_cm library lessens the workload of this procedure.

IBPP utilizes the rdma_cm api to establish and tear down connections. It creates a rdma_cm_id, which acts as a regular socket. In order to establish communication you then bind the remote address to the ID, and resolve it using rdma_cm library calls. After successfully connecting, the rdma_cm provides functions to create QPs, associate them with the remote host and transition them into a state where they are ready for sending. The details of using the API is explained in the RDMA aware programming user manual [42], along with example programs.

For comparison, the manual also provides a programming example for setting up connections using only traditional sockets and the verb calls. As seen, the amount of code required to set up connections using verbs only is significantly larger than what we need using the rdma_cm. Both creating the initial socket

connection to transmit init-data, and transitioning the QPs into a ready to send state has to be performed when using pure IB-verbs.

When the connection are established, all communication is performed through the creation and posting of work requests to the appropriate queues, both for sending and receiving messages. Work Completion Entries (WCEs) must be polled from the completion queue associated with the queue they were posted in.

Another feature introduced by the ibverbs library is the completion channel. This is an abstraction, and enables multiple completion queues to post notifications of completion to a single completion channel. When a notification is registered on the channel, the relevant completion queue is returned and the completed WCEs can be polled from a central processing thread. After each notification, the event must be acknowledged and notification must be re-requested using ibv_req_notify_cq.

3.2 Designing the IBPP framework

The IBPP library is a collection of classes, managers and threads which takes care of setting up and maintaining IB communication between end nodes. It requires some awareness of the IB programming model, but not to the same extent as working directly with the verbs-api. The main philosophy I had in mind when designing the framework was to abstract the components such as connections and operations to a level where the underlying IB-specific details become hidden from the user.

3.2.1 The IBManager

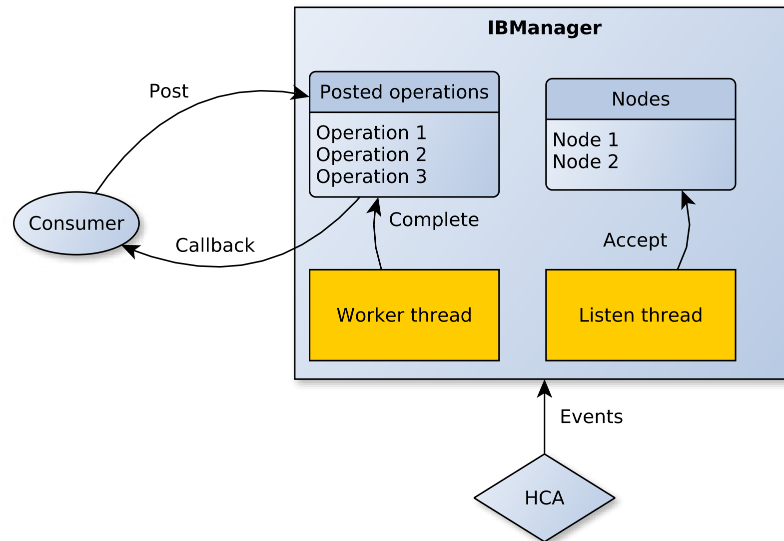

Figure 3.1: Consumer interaction with the IBManager

The core of the library is the IBManager class. It sets up a listener and a worker thread when it is instantiated. Figure 3.1 illustrates how they interact with the rest of the application. The listener thread listens for incoming connections on a specified port. When a connection is detected, it will be encapsulated in an IBNode class, and mapped to the LID of the connecting port for future reference. The worker thread processes elements from the CQ, and acknowledges CQ-Events on the completion channel.

The interaction between the consumer, the IBManager and the HCA can be seen in figure 3.1. Every operation that is created using the framework is registered with the manager. It is given a unique 64 bit Work Request ID (WRID), which is mapped to a pointer to the operation. When the worker thread registers an event on the CQ, it will poll all outstanding WCEs and verify that they were successful. A successful WCE will cause the associated operations virtual complete() function to be called. This allows the operation

to signal whether it want to be automatically deleted or not by returning a boolean value. Some operations, such as IBReceive, also have other tasks to perform upon completion, namely to push the received message onto its owners message queue. Asynchronous operations may also need to execute a callback function, to notify its owner of the completion.

For connecting to a node, the IBManager offers a connect-function, which takes the address and port as parameters. This function then connects to the specified node, and maps it for communication use at a later stage.

When both nodes are connected and have the IBManager instance running, the library offers a set of classes representing standard IB operations. As of now, RDMARead, RDMAWrite and RDMASwapCompare are exposed to the user, while IBSend and IBreceive are special cases used internally.

The send and receive-interface is used to send standard IB messages between nodes. To send a message, the IBNode class has a send member function. It provides two versions, one which takes a message ID and a data string as parameters, and the other takes a handle to an existing message to be transfered. For receiving, it provides a messageCount function, which returns the number of received messages in the queue, and a popMessage function which blocks and returns a shared pointer to a message handle containing the oldest received message on the node.

The RDMA-based operations are more directly exposed to the user, and they have to be created through the templated Operation::create-method, which has the signature as seen in listing 3.1. The create function returns a pointer to the created operation, which has to be explicitly deleted after its completion in the current iteration of the framework. The parameters describe the local and remote memory region to be affected by the operation,

Listing 3.1: Operation::create

```
template<class t_OP, class t_vARG>
Operation* Operation::create(Connection* conn, IBManager*
    man, RemoteDestination& local, RemoteDestination& remote, bool
    async, t_vARG* args=0,  boost::function<void()>* callback =
    NULL)
```

whether or not they are to be performed blocking or asynchronously and an optional operation-specific argument vector. The final argument is a pointer to a callback function which will be executed upon completion.

3.2.2 Memory management

Registering memory to be used for IB operations is relatively expensive. It involves translating the virtual address of the buffer to a physical address, and pinning the affected memory pages to avoid them being swapped out. As seen in [43], cycling pre-registered buffers increase the achievable bandwidth considerably. The factor of improvement for small buffers is as high as 3.22 times, while the relative cost drops for larger buffer sizes. Nevertheless, a key point of being able to utilize the potential of RDMA-operations is to avoid frequent registration of memory regions. Thus, having a pool of buffers registered at startup, and recycling those throughout the lifetime of the application is desirable.

The BufferManager class provided by the library, maintains a pool of template-specified objects that are registered with the HCA and ready for RDMA operations. It has a simple getBuffer function, which returns a shared pointer to a BufferManager::Handle instance, which in turn points to an object kept

in the buffer manager. When this handle is deleted, it will free the object and place it back in the pool, avoiding expensive registration/deregistration of the memory region.

In IBPP, a *RemoteDestination* is the equivalent of a pointer to an RDMA registered memory area. It specifies the LID of the node containing the buffer, and the address and size of the buffer in the node memory. Additionally, it contains the access information required for the memory region. All RDMA operations use *RemoteDestination*s to describe memory regions, both local and remote.

3.3 An example program

Listing 3.2 demonstrates how to write a simple ping-pong program using the IBPP framework. To illustrate the simplicity of using a framework such as this, rc_pingpong.c, which is part of the ibverbs example package, spans 781 code lines. It provides more or less the same functionality, with some additional features. The difference, however, is still clear.

Listing 3.2: A simple ping-pong program using the IBPP message interface

```
#include <iostream>
#include <string>
#include <IBPP/ibpp.hpp>

using namespace std;
using namespace IBPP;
using namespace boost;

int main ( int argc, char** argv )
{
```

```
try {
  int messages = 5;
  shared_ptr<BufferManager<IBMessage>::Handle> msg;
  shared_ptr<Configuration> config;

  // Instansiate the IBManager, passing the command line
  // arguments. The IBManager instance sets up a listener
  // and worker thread, and provides an interface for using
  // the IB fabric.
  shared_ptr<IBManager> manager = IBManager::create ( argv,
     argc );
  config = config;
  shared_ptr<IBNode> node;

  // Detect whether or not this node is the server
  // (extracted from the command line)
  bool isserver = config->count ("server") == 1;

  if ( !isserver ) {

    // The client connects to the specific address using
    // a simple call to manager->connect().
    int port = config->value<int> ("listenport");
    string address = config->value<string> ("address");
    manager->connect (  address, port );
    node = manager->nodes().begin()->second;

    // Send a messages
    for ( int a = 0; a < messages; a++ ) {
      node->sendMessage (52, "ping");

      // Wait for a response from the server
      while ( node->messageCount() == 0 )
        this_thread::yield();
      msg=node->popMessage();

      // Display the data from the message
      cout << msg->dataPointer()->msg << endl;
```

```
    }
  }
  else
  {
    // The server waits for a node to connect
    while ( manager->nodes().size() == 0 ) { }
    node = manager->nodes().begin()->second;
    for ( int a = 0; a < messages; a++ ) {

      // Wait for a message
      while ( node->messageCount() == 0 )
        this_thread::yield();
      msg = node->popMessage();
      cout << msg->dataPointer()->msg << endl;

      // And respond
      node->sendMessage ( 42, "pong" );
    }
  }
}
catch ( string& e ) {
  cout << e << endl;
}
catch ( ... ) {
  cout << "Caught unspecified exception, terminating." <<
      endl;
}
return 0;
}
```

3.4 Advanced usage and working with RDMA buffers

IBPP provides a set of classes that can be used to create buffers ready for RDMA operations. The easiest way to allocate a buffer, is to use the templated *BufferManager* class. It creates a pool of RDMABuffer objects of the templated type. Objects are obtained from the pool by calling its *getBuffer* function, which returns a shared pointer to a BufferManager::Handle instance. The handle contains the obtained buffer, and will place it back in the pool for re-use upon destruction.

The *RDMABuffer* class is a templated data storage class, which will allocate and register an array of objects for RDMA operations. The *RDMABuffer* has functions for retreiving specific elements of the array, or a pointer to where the data is stored.

The weakness of the provided buffer managers is that they do only support basic types, or types that require no constructor arguments. For instances where this is insufficient, the *RDMAObject* base class can be inherited to create customized RDMA capable classes. This requries overloading the virtual *void registerMemory(ibv_pd*)* function, which will be called upon construction. Implementing this function currently requires knowledge about the ibverbs library, and how to use *ibv_reg_mr* to register your memory. This is a design flaw, and should be fixed by creating templated helper functions to handle the registration before the library is released to the public.

In listing 3.3 we show a code sample of how the RDMA buffer related classes can be used for RDMA communication. This example assumes that a connection is established, that we have received the remote destination information

about the buffer we are accessing and that the lid of the node is known. The remote buffer information can be exchanged by the use of send/receive operations, or by utilizing a *RDMABuffer<char>* made available for all connections to exchange information after they have been established. The RemoteDestination of this buffer can be obtained from the *Connection* class' *remoteDest()* function.

Listing 3.3: Using the RDMA buffer classes for RDMA operations

```
// Extract the connection to the node
// storing the remote buffer
Connection* connection=ibmanager->node
    (remoteNodeLID)->connectionPtr();

// Create a buffer manager with 10 int64_t buffers
BufferManager<int64_t> bufferManager(ibmanager,10);

// Create a handle and retrieve a buffer to use
boost::shared_ptr<BufferManager<int64_t>::Handle> handle;
handle=bufferManager->getBuffer();

// Retrieve the remote destination of the object to read
RemoteDestination remdest=getRemoteObjectPointer();
RemoteDestination localdest=handle->destination();

// Read the remote object into our local buffer using
// a blocking RDMARead operation
delete Operation::create<RDMARead,
    uint64_t>(connection,ibmanager,localdest,remdest,false,NULL);

// Print the newly read value
cout << (*handle->dataPointer()) << endl;
```

3.5 What remains to be done

In its current state, the IBPP framework is in a working condition, as in being able to facilitate communication between nodes on an InfiniBand network. Much work remains to be done in order to make it a practically usable product in real world applications, though. It requires optimization to bring down the latencies of operations even more, and the consistency of the class design is lacking at best. As it stands at the time of the writing of this document, the IBPP serves only the function that is required of it to enable the experimental implementation of the DVM.

Given time and development, it should be possible to turn the IBPP into a feasible and easily accessible C++ framework for the verbs API. By establishing a standardized interface, one could prevent conflicts occurring due to the lack of standardization of the underlying API. Although the Open Fabrics Alliance are providing a more or less stable software stack, it can become subject to change in future releases. The library will have to be updated in accordance with the new specifications, but it should be possible to maintain backwards compability by abstraction of the provided functionality.

3.6 IBPP performance evaluation

As the DVM relies on the IBPP framework for all communication, it is important that we properly analyze the performance of the library to ascertain that the underlying operations perform satisfactory. Ultimately, we want the IBPP to perform at the same level as a pure VAPI-implementation. We performed a number micro benchmarks to discover the properties of the various

operations that has been implemented, and compared them to the benchmarks provided by the OFED stack. The source code for all the benchmarks is available in the OFED source distribution.

The microbenchmarks mainly focus on the latency and achieved bandwidth of our operations, when compared to the perftest package, for various message sizes.

Our test environment is a small network of nodes using Sun 4xDDR Dual Port HCAs, connected to a SilverStorm 9024 DDR switch. The nodes have 2gb system memory, and an AMD Opteron 2210 dual core processor. The cluster is running a rocks distributed CentOS kernel version 2.6.18-164.6.1.el5 operating system.

3.6.1 Memory registration performance

As discussed in [28], registering memory regions to be used for RDMA operations is a relatively expensive operation. In the IBPP, the BufferManager maintains a pool of pre-registered buffers that is circulated to prevent the overhead associated with registering the required buffers during runtime. Nonetheless, it may often be necessary to de-allocate buffers during runtime, since the memory pages that the buffers belong to needs to be pinned as long as the registration persists.

When investigating the overhead created by the RDMABuffer class, I measured the time taken to allocate and register the memory for 1k buffers of increasing size, the cost of de-registering those buffers and the comparative cost of locally allocating an array of the same size as the RDMABuffer.

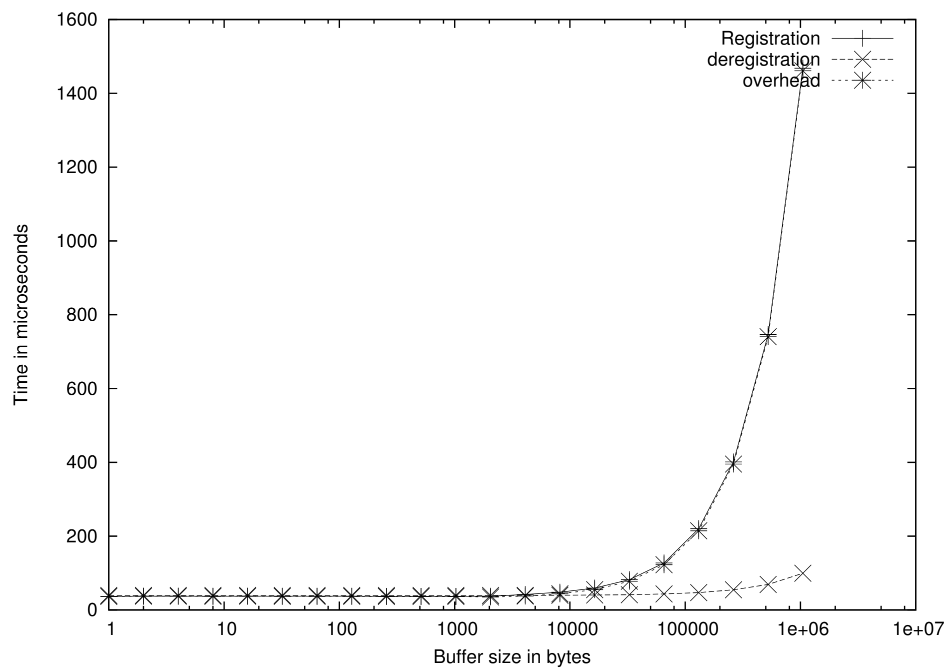

Figure 3.2: Cost of allocating and registering RDMABuffer objects, overhead and de-registration cost

The results of the analysis can be seen in figure 3.2. The conclusion is in line with what was discovered in [28]. For buffer sizes up to the page size of the system, which for the test setup was 4096 bytes, the registration cost is more or less constant. When buffers start exceeding the page size, the overhead starts growing linearly with the size of the registered MR, and the overhead compared to local allocations follows the same curve. De-registration is a lot cheaper than registering, and grows only slightly with the buffer size. In conclusion, having buffers being registered at the time they are needed greatly impacts the performance, as DMA-registration is expensive. We want to use recycled buffers, and IBPP offers tools to do this.

3.6.2 Send and Receive latency

Now it is time to have a look at the send and receive functionality of IBPP. To benchmark the performance of the framework, I used the ibv_rc_pingpong which is part of the perftest package in the OFED stack. Figure 3.3 shows the Round Trip Time (RTT) versus message size for both IBPP and the benchmark. Every test was ran over 10k iterations, and the message sizes range from 256 to 32768 bytes.

As demonstrated in the figure, IBPP seems to have an additional constant overhead of around 25μs compared to the benchmark. This is mainly due to the added delay of having to wait for the receive-request to be completed in the worker thread before being able to respond. I believe that improving the performance of those operations is possible, by changing how these particular operations are handled in the framework. Specifically, having them take a shorter path through the registration process will be greatly beneficial. However, as it was not critical for the testing I have been doing during this thesis, I have not spent much time on optimizing the send and receive

operations. It will have to be improved drastically for the library to be practically useable in the future.

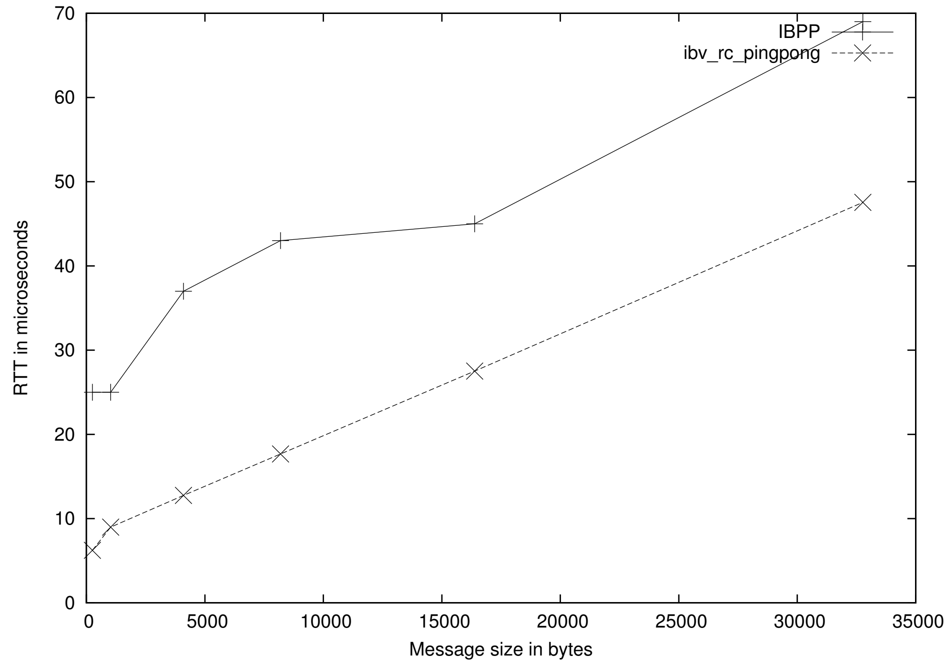

Figure 3.3: RTT versus message size

Message size	IBPP	ibv_rc_pingpong
256	136.53	656.93
1024	528.51	1824.74
4096	1638.40	5136.77
8192	2912.71	7414.87
16384	4854.51	9524.96
32768	6636.55	11028.26

Table 3.1: Achieved bandwidth in Mbit/s through message passing

3.6.3 RDMA Read

Figure 3.4 reveals that the RDMA Read operation of IBPP lies much closer to the benchmark programs than the send/receive functionality does. The difference in latency starts out at around 6μs at the smallest message sizes. For sizes up to 256 bytes, the framework has a constant latency of roughly 8μs. For 64 byte messages, the benchmark is 3.442 times faster, which is obviously much worse than desired. For the important 65k , which is the page size used for the DVM, which will be discussed later, it is only 1.121 times faster, and at 1MB buffers, the factor is 1.002.

Table 3.2 shows the achieved bandwidth using blocking RDMA Read operations with the specified buffer sizes. Both ib_read_lat and the IBPP

Buffer size	IBPP	ib_read_lat
256	29.33	92.08
4096	314.74	577.71
65536	1312.16	1470.73
1048576	1624.85	1628.83

Table 3.2: Achieved bandwidth in MB/s through blocking RDMA reads

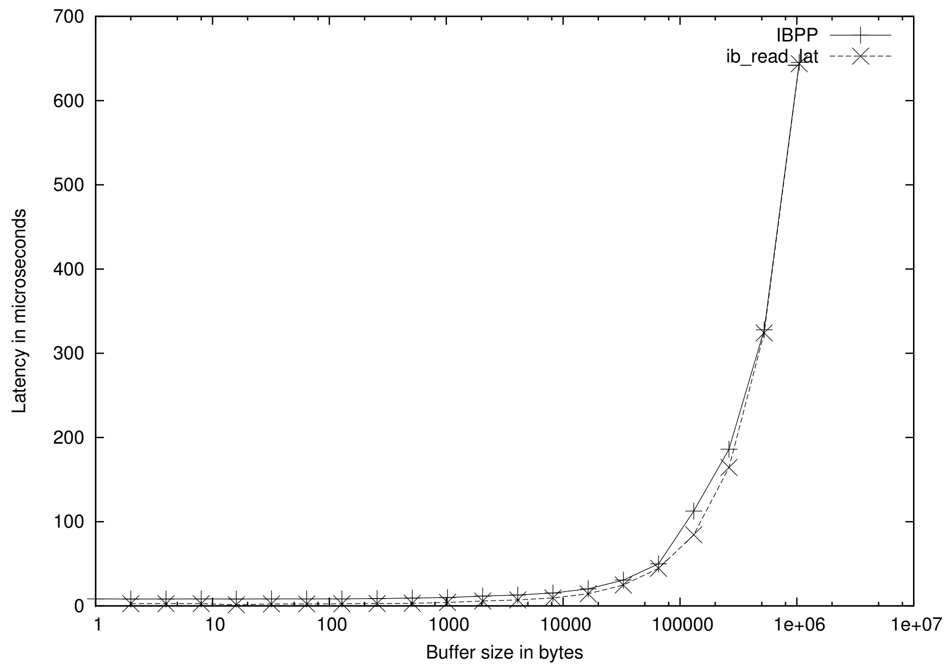

Figure 3.4: Read latency versus buffer size

operations used for this test use blocking synchronous read operations. The achievable bandwidth for the smaller blocking read operations is fairly low, as would be expected. Since the send queue will only receive one operation at the time, the capabilities of the HCA are not fully exploited. The next section examines the maximum achievable bandwidth, using asynchronous reads, resulting in the hardware being used more efficiently for smaller messages.

Figure 3.5 shows the achievable bandwidth for asynchronous operations. The IBPP test application creates a RDMABuffer array filled with buffers of the desired message size on the passive side, and cycles asynchronous read operations on them. It then waits for all the outstanding work requests to be completed before timing the entire loop. ib_read_bw uses a tighter loop, filling up the SQ with read operations, and then waiting for the batch to be completed until n iterations are complete.

Buffer size	Asynchronous latency	Synchronized latency	Factor of Improvement
256	2.195	8.726	3.97
4096	5.791	13.014	2.25
65536	43.939	49.945	1.14
1048576	656.258	645.34	0.98

Table 3.3: IBPP read latencies for asynchronous and synchronous operations

When looking at table 3.3, we see that the per-operation time cost of performing smaller reads improves drastically when doing asynchronous reads. The factor of improvement for packets of size 256 bytes is as high as 3.97. This is to be expected since it removes much of the overhead of waiting for a completion before performing the next read, but it is important to be aware of this behavior. When dealing with smaller transfers, it is a good idea to perform them in bulks of asynchronous read operations rather than perform a blocking read.

There are, of course, instances where asynchronous reads are undesirable, but for cases like reading memory pages it would be wise to acquire all the relevant locks first, and then performing the reads in bulk collectively. For very large buffer sizes, where we utilize the network more fully, the performance actually drops when compared to doing one read at the time. This behavior is also seen in ib_read_bw, where the bandwidth for 1MB buffers is 1564.39MB/s for asynchronous reads, and 1628.83MB/s for ib_read_lat.

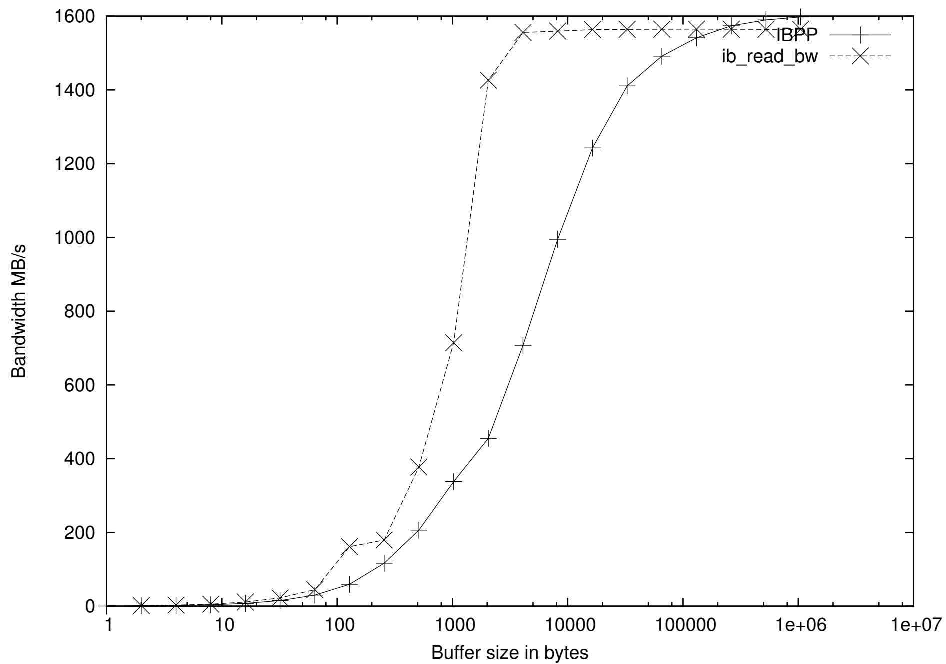

Figure 3.5: Maximum achieved bandwidth versus buffer size

3.6.4 RDMA Write

Figure 3.6 shows us that the RDMAWrite operation has more or less the same performance in comparison to the ib_write_lat application as RDMARead had to ib_read_lat. The graph for achieved asynchronous write bandwidth is more interesting, as figure 3.7 shows. The asynchronous write operation actually surpasses the benchmark around the 64kB mark. This might be a side-effect of the test application using multiple write buffers rather than one, providing slightly higher local memory throughput, which starts to impact the result for larger buffers.

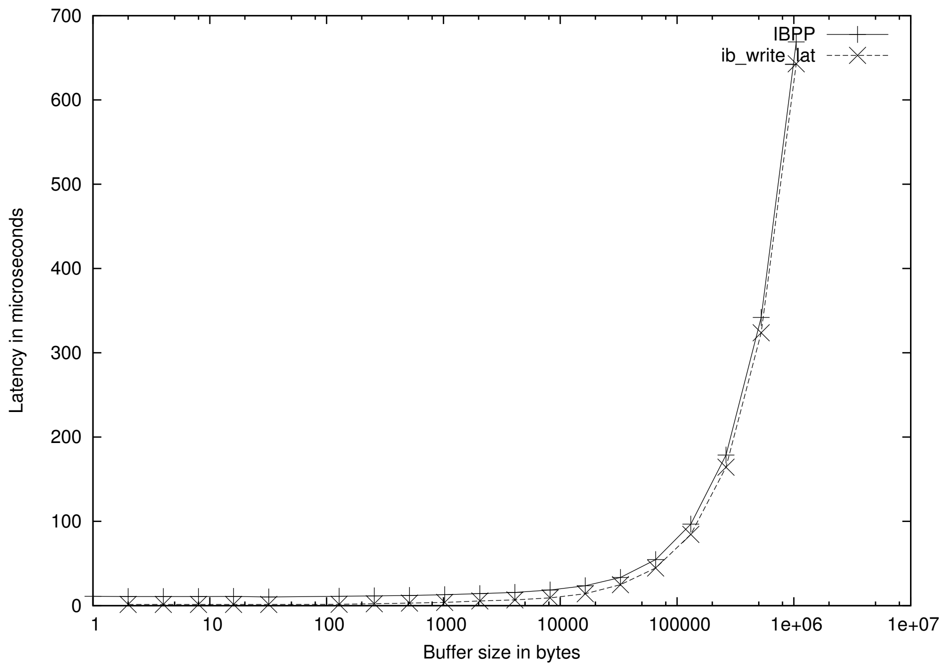

Figure 3.6: Write latency versus buffer size

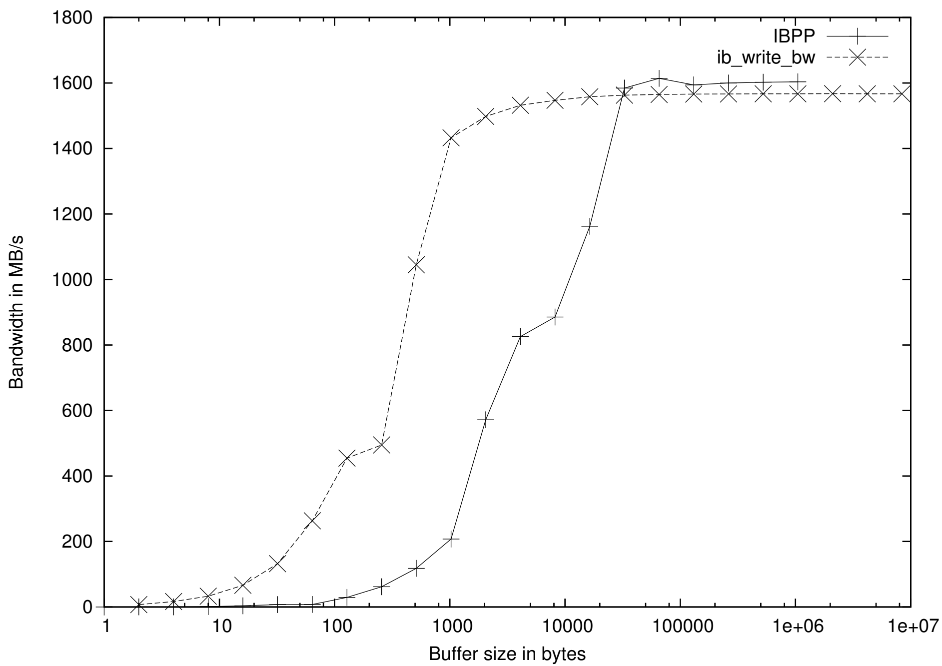

Figure 3.7: Asynchronous write bandwidth versus buffer size

3.6.5 RDMA Swap-Compare evaluation

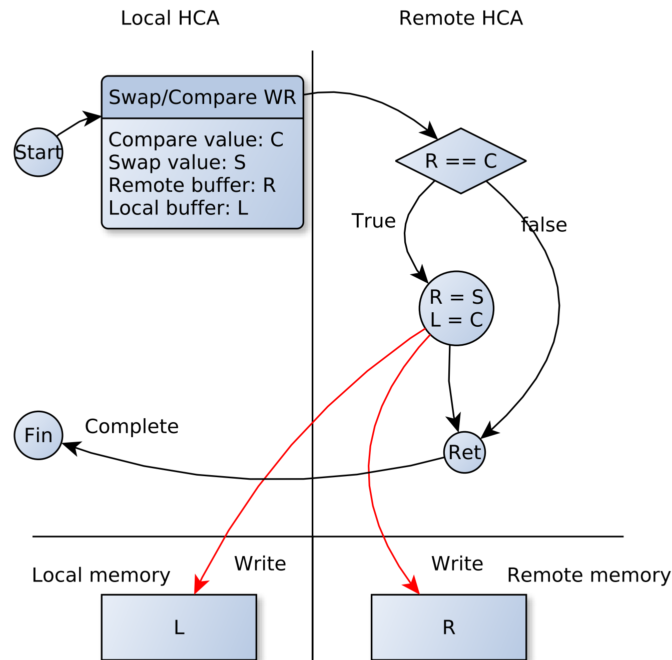

Figure 3.8: Atomic swap/compare access flow

The atomic swap-compare operation allows for atomically checking a remote 64 bit variable. The value of the remote buffer is checked against a specified value, and if they match, the value is swapped with that of a local buffer, effectively providing a RDMA based mutex operation. Figure 3.8 shows how the atomic swap compare operation is executed in the remote HCA. Most importantly, all the logic is performed within the HCA itself, and no kernel interrupt is used to access the atomic variable.

This functionality allows for a very straight forward approach to synchronization when compared to conventional distributed locking schemes. The atomicity of the swap compare-operation allows us to use a remote buffer

as a spinlock without the need of a centralized lock manager and control messages beyond the read operation.

The swap/compare-operation has a cost of approximately 9.75μs on our test cluster. As there is no test for this feature provided with the perftest package, it is difficult to assess how this portion of the library compares with a benchmark on our cluster. The tests performed in [40] measures the atomic operations around 9μs. For comparison, a local mutex lock using the boost::mutex library takes about 0.031μs. This means that acquiring a remote mutex lock is 300 times slower than a local lock. It is also worth noting that a conventional network locking mechanism would require at least two messages to be sent. ibv_rc_pingpong achieves a RTT of 5.46μs with a message size of 8 bytes, i.e half the cost of a swap/compare operation. When processing time on the lock manager is taken into account, it is not an unreasonably expensive operation. The atomic IB operations provide a cheap way of protecting certain critical areas without devoting processing time to a lock manager, and indeed without even interrupting the kernel.

Swap/compare does not, however, provide any fairness property for lock acquisition in the form of access queues. Additionally, swap/compare has to spin lock when a mutex is unavailable, causing more load to be put on the fabric and the node storing the lock. This is partially solved by using an exponential pull-back, as proposed in [34], but this can lead to starvation. In the future, implementing a more robust locking scheme as the one described in [25] is a priority. Expanding IBPP with a more advanced and robust synchronization system will be important for future usage.

Chapter 4

The Distributed Virtual Machine

The virtual machine is built up by several logical layers of execution. The performance and inner workings of these layers is discussed in detail later in this document. Here is a brief explanation designed to provide a basic overview of the DVM.

4.1 Architecture

Figure 4.1 shows the general architecture of the DVM. It consists of processing nodes, which execute the Distributed Assembly (DASS) application, and paging nodes, whose responsibility is to maintain the global shared memory

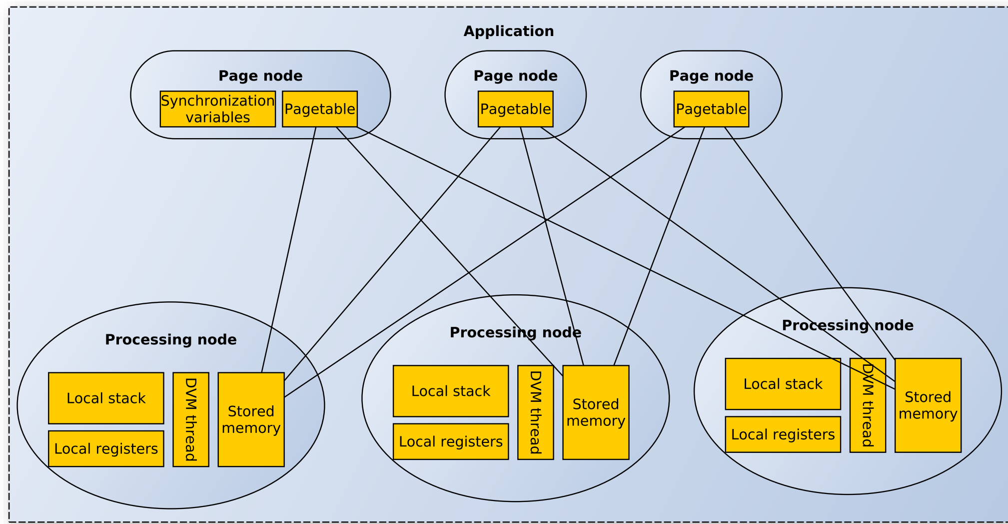

Figure 4.1: Architecture of the DVM

mapping. A subset of the page nodes is also tasked with the responsibility of storing the synchronization variables used in the bytecode.

4.1.1 The instruction set

The instruction set for the Distributed Assembly Language (DASS) is loosely based upon the x86 instruction set [32]. It is currently capable of simple arithmetic, memory and flow control operations. It has no floating point arithmetic, and it lacks the synchronization and threading operations required to create real life applications for the DVM.

The syntax of the DASS language is fairly straight forward, and resembles that of x86 with a few twists and turns. The basic syntax forms are shown in listing 4.2. Instruction operands can be either register numbers, numerical values, jump points or array access. In its current implementation, the DVM

does not support array access to registers. The different operand syntax forms are displayed in listing 4.3.

```
; Commented lines starts with a ";"
instruction operand1 ... operandN
:jumpLabel
```

Figure 4.2: DASS basic syntax forms

```
; Adding the numerical value 1 to register 0
add 0 $1

; Adding the value stored in register 1 to register 0
add 0 1

; Adding the value stored in register 1 to register 0
set 2 $0
set 3 $1
add [2] [3]

; Jump to the start label
:start
jmp start
```

Figure 4.3: DASS operand syntax

The operations of the DASS instruction set can be split into 4 main categories. Memory instructions, register operations, control instructions and threading related instructions. These are general classifications however, and they have overlapping characteristics. A full listing of the instruction set can be found in appendix A. We will discuss the future of DASS and how it impacts the performance of the DVM in section 5.3.

4.1.2 The application

The application is the layer which manages the virtual shared memory space of the DVM. Loading and starting the execution of the bytecode is also the responsibility of the application. It stores the attributes of the application being run, such as the number of registers, stack size and the instruction set that is being executed. It is also responsible for swapping in pages that are not locally present, and keeping track of the currently owned page leases.

4.1.3 The memory architecture of the DVM runtime system

The shared memory of the DVM uses a 64 bit address space. An address within this space is split into two parts, one for the page ID and one for the page offset. When the application receives a page request from a thread, it decodes the address and extracts the page ID using a bitmask. The page ID is then used to directly look up the page node responsible for that page.

4.1.4 The runtime environment

Each thread running inside the virtual machine is given its own set of registers and a local stack. The amount of registers available, the stack size and other properties of the runtime is customizable by the application definition. The thread is represented by the Runtime class, which takes care of initializing the stack, registers, counters and everything else associated with interpreting the program code.

The stack provided for the threads is local-only. It lies outside of the shared address space of the DVM, and the implications of this is discussed more thoroughly in chapter 5. There we discuss various details of the implementation more closely.

4.1.5 The dispatcher

The dispatch thread is started by the Runtime class, and it is the core executional unit of the DVM. It fetches instructions from the code and executes them. When accessing the shared memory, the dispatcher does not consider the shared aspect of the memory model, beyond requesting that a page is present at the time of access. The page swapping and leasing is handled by the Application unit.

4.1.6 The page node

Figure 4.4 shows how the memory pages in the memory space are mapped to and stored on the different computing nodes. Every page node is responsible for maintaining an up-to-date page table for a limited window of the shared memory space. The page table consist of PagetableEntry objects, each of which is an RDMA object accessible by RDMA operations. The page table entry consist of a 64 bit access lock, a flag containing the current page state and a RemoteDestination entry. The access lock is used for atomic swap/-compare operations, effectively acting as a mutex for the page state. When a node wishes to set the locked state, it needs to own the mutex. Unsetting the flag, and thus releasing ownership of the page, does not require locking.

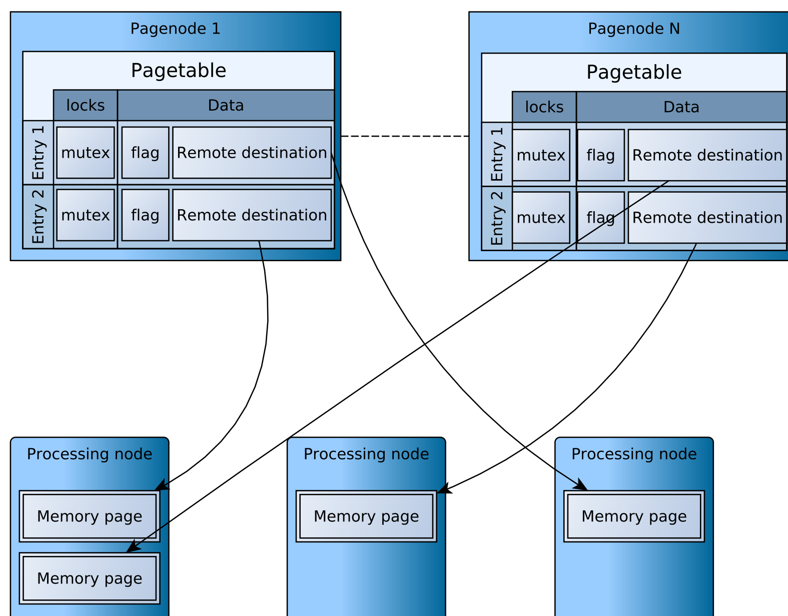

Figure 4.4: Memory storage model for the DVM

Finally, the remote destination specifies the node currently owning the page as well as pointing to the buffer where the page data is stored on the owner.

When an application wants to access a memory page, it waits for the locked flag to become unset. It then attains ownership of the page by setting the flag, and reads the rest of the entry using an RDMARead. After acquiring the node, it reads the page from its owner, and writes the new remote destination to the pagenode responsible for the page. It also sends a swap-out message to the previous owner of the page, allowing it to free its buffer to be reused. Finally, it creates a lease for the page, which ensures ownership of the page for a period of time. The algorithm for acquiring page ownership is visualized in figure 4.5. Single-headed red arrows indicate RDMA operations, dual-headed indicate an atomic swap and compare. As we can see, the algorithm starts at *1* by waiting for the lock to become available, and acquiring it. It then proceeds to *2*, where it does consecutive reads on the flag until the page is no

longer in use, upon which *3* will read the data from the remote location into a local buffer. *4* updates the pagetable entry to point to the new location of the page, before *5* finally releases the mutex lock.

For the lifespan of the lease, it keeps the pages local flag set to true. As long as this flag remains set, a runtime accessing the page only has to perform a local comparison in order to access the memory. When the lease expires, it releases the flag, and writes it to the pagenode. For subsequent access, the page table must be queried before accessing the memory again.

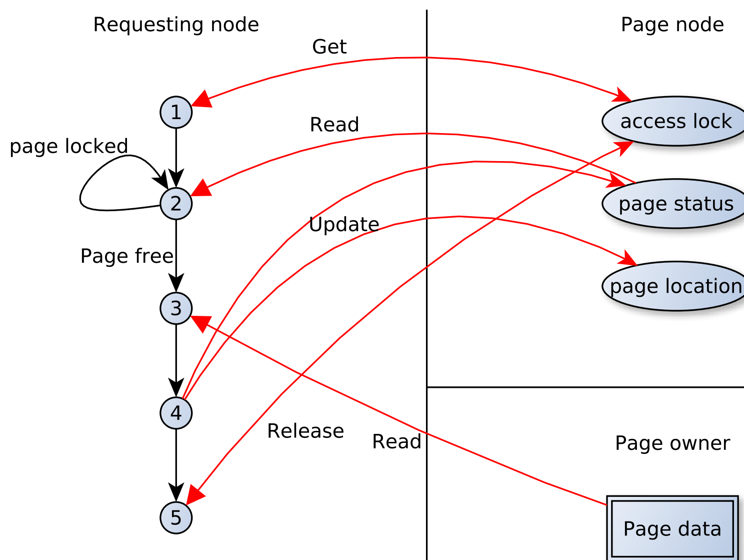

Figure 4.5: Page swapping algorithm

4.1.7 The computing node

The computing node runs on all the processing nodes connected to the machine. It sets up the IBManager, reads the application directory and connects

to the rest of the DVM. It maps pagenodes to addresses, and establishes communication between the computing nodes. It instantiates and loads the application, and also processes control messages such as swap-out requests and error handling.

When instansiated, the node spawns a worker thread, which takes care of releasing expired page leases. When a page lease expires, the thread will block until the page is no longer in use locally, and release the locked flag in the pagetable entry of the page in question.

4.1.8 The network topology file

Currently, the DVM uses a rudimentary topology file, describing the nodes to be used and their functionality. It specifies the window of memory each pagenode is responsible for, what dass files to load on each computing node and the connections to be established between computing nodes. This is obviously a bare-bones way of making the testing feasible, but with a few extensions and tweaks the topology file is an excellent static way of describing the physical layout of the DVM. It will also be possible to generate revised versions of the topology files based on runtime observations and profiling, to maximise utilization of the network. This is briefly discussed in section 5.10, as a possible future optimization.

The applications in the DVM are represented by a folder, containing the nodes.txt files, and all relevant .dass files to be executed by the application. Each line of the node topology file is either a key specifier, denoted by *[keyname]* or a node beloning to the last specified key. The compute node description is on the form *LID ip [connect count] [list of nodes to connect*

to], and the page node lines have the from *LID ip memory-mapping*. The connect count specifies the number of nodes the compute node will establish connections to, and the nodes are specified in the following list. For the page nodes, the memory mapping designates which area of the memory the node is responsible for. A positive memory-mapping value tells the node that it is responsible for the memory pages *[memory-mapping * pages per node]* to *[memory-mapping * pages per node + pages per node]*. A negative value describes an index from *[MAX_MEMORY - pages per node * memory mapping]*.

Listing 4.1: An example node topology file

```
[pagenodes]
6 25.1.1.5 0
7 25.1.1.6 -1
[computenodes]
9 25.1.1.8 store.dass 1 25.1.1.7
8 25.1.1.7 loop.dass 0
```

4.2 DVM performance

It is now time to have an in-depth look at the performance of the DVM. We perform a number of micro benchmarks to assess the relative speed of the various operations in comparison to C++, as well as analyzing small test applications to take a look at the performance of the DSM system we have implemented. The most interesting properties we look at are memory access bandwidth for both local-only and contested memory, and the cost of page misses.

4.2.1 Register-to-register performance

For determining the performance of the DVM, it is important to thoroughly analyze the speed of the non-memory related operations of the VM. I have performed a number of tests of the basic register operations in comparison to compiled code combined with inline assembly optimizations to get a grasp of the relative execution speed.

The first test I performed was comparing a simple loop with addition in the DVM to the equivalent C++/assembly compiled version. It increases the loop variable n and adds it to a summation variable each iteration. The test code can be seen in listings 4.2 and 4.3. As observed in table 4.1, the speed is 8.81 times slower than what the compiler generated binary runs at.

I also performed a test using a volatile sum-variable in C++, to simulate the circumstance where the processor cache is exhausted, and the access is forced to be performed on the stack. Although this is not a highly likely scenario in and of itself, the theoretical large number of registers could allow it to become relevant under certain circumstances. The result shows that the DVM is about 9 times slower at accessing variables on the stack when compared to C.

This puts the DVM at a basic instruction overhead between 8 and 9. Given that it lacks any form of advanced optimization such as JIT, the performance is satisfactory for the testing we are doing in this thesis, since we will mainly be looking at the memory related operations.

Test	DVM	C++	Cost factor
Iteration with addition	14690μs	1667μs	8.81
Addition of volatile variables	41046μs	4418μs	9.29

Table 4.1: DVM and C++ speed comparison over 1000000 iterations

Listing 4.2: DASS iteration speed test

```
set    0 $0      ; a = 0
set    1 $0      ; n = 0
set    4 $1000000
out    0
clk    2         ; t0 = now
:loop

  add    0 1
  for    1 4 loop

clk    3         ; tn = now
sub    3 2       ; tn -= t0
out    3         ; cout << tn
hlt
```

Listing 4.3: C++ iteration speed test

```
uint64_t n = 0, a = 0;
size_t N = 1000000;
while ( n< N ) {
  __asm__ ( "addl %2 %1"
        :"=r" (a)
        :"r"(n), "r"(a)
        :"0" );
  __asm__ ( "inc  %1"
        :"=r" (n)
        :"r"(n)
        :"0" );
}
```

4.2.2 Push and pop performance

Now it is time to have a look at the push/pop performance of the DVM. These operations have the possibility of processing multiple registers in bulk on the DVM, as opposed to x86 assembly counterpart, which only acts on one register at a time. As processing multiple registers in one operation removes the overhead of calling the instruction several times from the bytecode, I tested the speed of both versions up against the inline assembly counterpart. The results can be seen in figure 4.6.

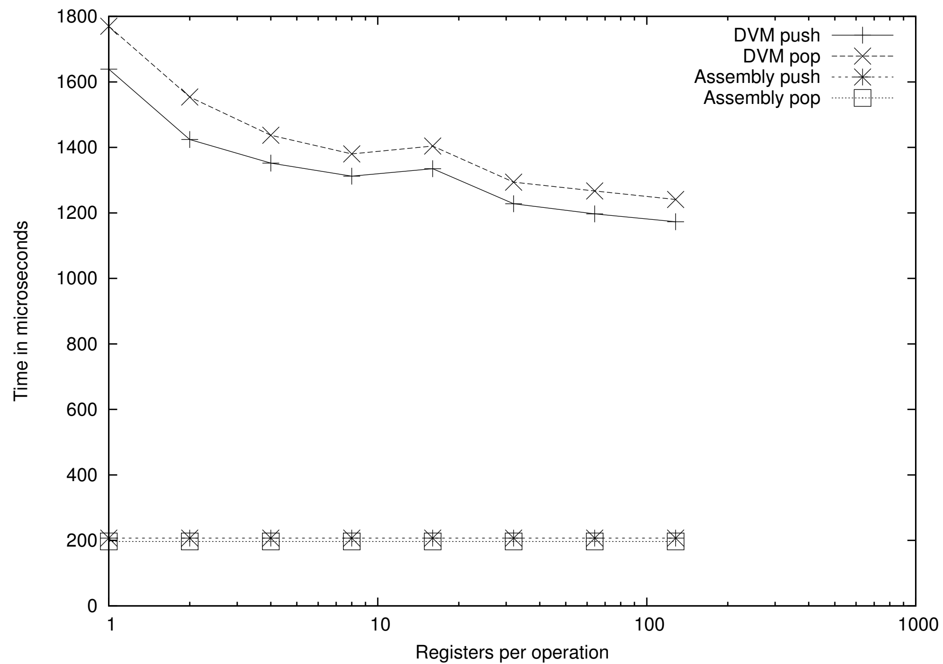

Figure 4.6: Push and pop comparison for 1MB of data

The test applications pushed and popped 1MB on the stack in 16 bursts of 64kB each. Each burst was timed and summed up to get the results provided here. The DVM test application utilizes the ability to manipulate a range of registers in a single operation using push and pop. At a register count of 1, they behave identically to their x86 counterparts, and the higher the range of registers being pushed is, the lower the VM-imposed overhead per register is.

As we can see, the bulk push/pop is actually fairly fast when compared to pure assembly. When using 1 register at the time, the push operation costs 7.91 times more than an assembly push, and the pop operation is 8.98 times slower. At the 128 register mark, the cost factors are 5.66 and 6.29 for push and pop respectively.

4.2.3 Page locking and swapping

When a memory operation is executed from the bytecode, the runtime will request the destination memory page from the application. The application will then first check to see if the "local" flag of the memory page is set. This flag is directly related to the current state of the page, i.e if it is set, we are guaranteed that the page is currently marked as locked in the page table, and it is safe to assume ownership for the duration of the operation, or until the "in use" flag is set to false.

If a page is not currently owned, the application will proceed to query the page table for the current location of the memory page in question. There are three possible states a non-owned page can be in, namely undefined, local but not owned and remote.

An undefined page has not previously been accessed by the application, and has no associated node in its page table entry. When reading an undefined status from the pagenode responsible for the requested address, an application will acquire the page lock, get a free page buffer from the system and write its remote destination to the page table, effectively registering the page in the virtual machine and making it available for the shared memory space.

A local but not owned page resides in the local memory of the current node, but the access lock will be freed. When this happens, the node will simply acquire the lock and return.

Finally, a non-local page will have to be locked, read and have its page table entry updated to the new physical location of the page buffer used. This operation has an extreme cost when compared to the other two alternatives,

State	Cost
undefined	90.5μs
local but not owned	43.3μs
remote	1500μs

Table 4.2: Cost for acquiring ownership of a page for the three states under race conditions

since the current owner of a page has a tendency of prolonging its ownership during heavy memory access. This is mainly caused by the rate at which new instructions are fetched and executed; the used flag is simply set at a higher time-resolution than what the thread responsible of releasing the memory can respond to.

As demonstrated in figure 4.2, a full swap is extremely expensive. Most of the cost comes from spinning on the lock flag, waiting for it to become available. This test is not, however, especially realistic when compared to a real world scenario. Normally, access to shared memory regions such as this will be synchronized using separate synchronization mechanisms, which does not rely on the shared memory. As most of the cost of the remote swap in this test comes from waiting for the page to become available, proper synchronization will remove most of the overhead from this, since it will remove the race conditions involved.

4.2.4 Shared memory access

When evaluating the shared memory access in the DVM, there are several metrics we need to consider. Both achievable bandwidth and latency needs to be measured and compared to its native code counterpart. Additionally,

we need to compare the performance for the case with no page swapping to when there are several processes competing for the same memory page.

The first test compares the store operation for N registers to memory to a basic C memcpy for the same buffer size. The results of this test are shown in figure 4.7. In this test, there is no swapping of data from remote nodes, but ownership of the page has to be acquired from the pagenode every time the page lease expires. The graphs *DVM store* and *DVM load* shows the performance of the uncontested memory access. As expected, the overhead of the DVM makes small copies much less efficient than memcpy, but it is interesting to observe that at 512 byte copies and up, the pure C copy is 4.49 times faster, and at 1024 bytes per copy it is 2.56 times faster. For the tests performed here I used a lease time of 100μs. The impact lease time has on performance is discussed and evaluated in chapter 5.

Test number two examines the impact a second node accessing the same memory region has on the store test. The result of this benchmark is denoted by the graph *store with competition* in figure 4.7. The test code for the measuring is displayed in listing 4.4, and the polling code, which is responsible for generating competition for the memory area, is in listing 4.5. As we clearly see, the achievable bandwidth drops significantly when there is constant competition for the memory pages. It is also worth noting that the results vary greatly, depending on access order, how long each page lease gets locally extended and other semi-random runtime factors such as thread scheduling and network load.

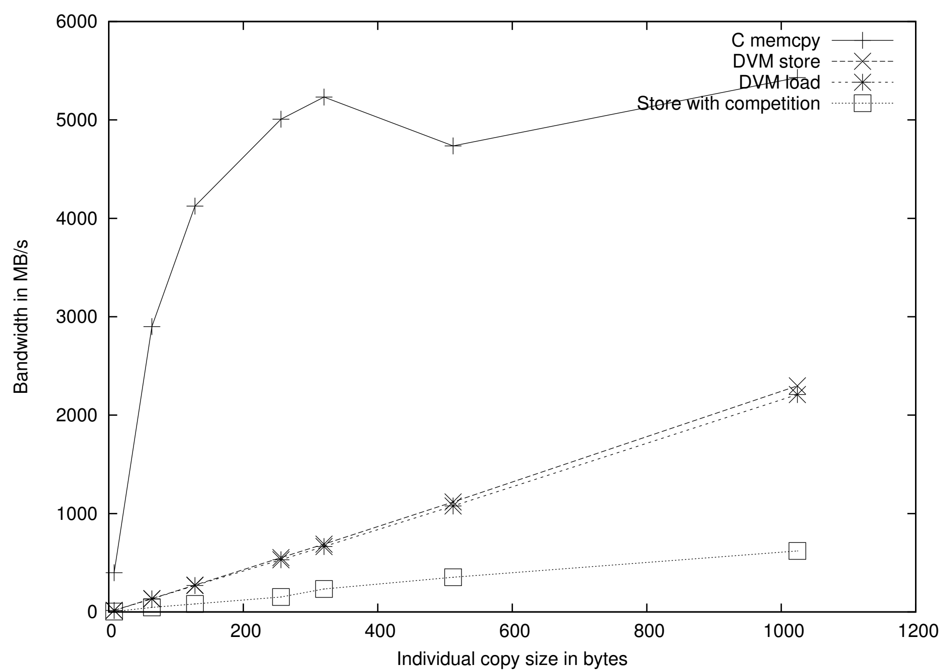

Figure 4.7: Transfering 10MB using DVM memory operations and C memcpy

Listing 4.4: Store performance test

```
set     0 $0
set     150 $500
set     5 $24

;wait for the init variable to
;be altered at memory address 24
:startloop
  lod 5 0 $1
  cjl 0 150 startloop

set     0 $0
set     150 $0        ; n = 0
str     0 0 $1
set     5 $1310720
str     0 1 $1
set     1 $42
clk     2         ; t0 = now
:loop
  ;Store 1 registers starting
  ;at 1 to the address stored
  ;in 0
  str     0 1   $1
  for     150 5 loop

clk   3      ; tn = now
sub   3 2    ; tn -= t0

set   120 $10485760
div   120 3
out   120

;Notify the looping thread
;that we are done
set 1 $1000
set   0   $0
str 0 1 $1

hlt
```

Listing 4.5: Page polling thread

```
set 0 $0
set 1 $0
set 2 $743
set 0 $24
; Write the start signal
str 0 2 $1
set 0 $0
:loop

  lod 0 1 $1
  ; As long as the program is
     running
  cjl 1 2 loop
hlt
```

Chapter 5

Discussion

Having introduced and analyzed the performance of the various components of the IBPP framework and the DVM, it is now time to discuss and evaluate them in more detail. We take a look at various optimization techniques used, and explore other possible solutions and improvements that can be made in the future. It is also important to look at a few implementation specific details, and to isolate problems that will need to be addressed in the future.

5.1 The dispatcher

There are a variety of approaches to interpreter implementation. Here we discuss a few of them, and explain the method chosen for the DVM.

The dispatcher of the DVM uses a direct threading model. It removes the

lookup-overhead from indirect threading by storing the physical address in the instruction struct.

Pre compilation is done at startup by iterating through the instruction list, and storing the next possible instruction addressees. This technique requires GNU C, as discussed in [24], since ANSI C does not support storing the address of jump points as pointers, which is required for threaded dispatch. As we only target platforms which support GCC for the DVM, this should not be an issue.

During runtime, the only operation required for the fetch/execute cycle is to increase the instruction pointer, and jumping directly to the address of the new instruction. This model of interpretation benefits greatly from branch prediction in modern processors [26]. Additionally, the Opteron 2210 processor used on our test cluster has a relatively short pipeline, punishing branch mispredictions far less than a processor with a longer pipeline would. This makes the DVM dispatcher fairly fast in our testing environment, as the performance analysis shows.

There exist a few other options for the dispatcher than direct dispatch. The simplest of those is to use a switch-statement for the operation type. The overhead imposed by the statement is quite large, as it requires an additional jump back to the start of the block after every fetch. Additionally, it removes most of the benefit gained from branch prediction, and [26] shows that threaded code is almost twice as fast on processors with efficient branch prediction. This approach is thus only appropriate when the development is limited to ANSI C.

Pure direct threaded code, as discussed by Bell in [19], stores the address of the next operation directly in the instruction list, along with the operands

Listing 5.1: Switch-based dispatch

```
ins = &instructions[0];
while ( running ) {

  switch ( (*ins)->type ) {

    case INS_PUSH:
      stack[sp--] = (*ins)->op2;
    break;
  }
  ins++;
}
```

Listing 5.2: Direct threading

```
ins = &instructions[0];
(*ins)->location = &&INS_PUSH;
goto (*ins)->location;

INS_PUSH:
  stack[sp--] = ins->op2;
  goto (*++ins)->location;
```

to the instructions. Listing 5.2 shows pseudo code of the direct threading approach used in the DVM.

To make the DVM viable for heavy computational tasks, using an interpreter is insufficient. Even though the dispatcher is fairly fast for being an interpreter, just-in-time[15] compilation is needed in order for it to be a feasible computing platform in the future.

5.2 The runtime environment

The runtime environment of the DVM is highly flexible when it comes to application preferences. The number of available register, the size of the shared memory, the memory page size and the stack size are all customizable. This leaves defining language-specific runtime systems up to the compiler implementation, and experimentation on the most efficient solutions will have to be performed in the future.

5.3 Distributed assembly

Currently, the DASS files are loaded from text into the DVM when starting an application. Writing an assembler to pre compile the assembly files to bytecode would decrease load times for larger applications.

Expanding the DASS format to include JIT-metadata is also a possible improvement. This would enable compilers to mark procedures and sections of the code that are suitable for JIT-compilation at compile-time, decreasing the profiling load during runtime.

A lot of work can be done to optimize the DASS language. Merging common operations into superoperators, as proposed in [47] can drastically reduce the dispatch overhead for certain common operations. This is partially done in the DASS for operations such as *for*, combining a compare, increase and conditional jump into one instruction. The ability to process multiple registers in singular memory operations also fall under this category of optimization.

5.4 The shared memory model

The shared memory of the DVM currently only supports full read/write access locking. This causes accessing memory under race conditions to be very slow, but this problem can be partially circumvented by the use of proper synchronization mechanisms to control access. This is mainly because of the fact that the biggest cost factor when acquiring remote memory is waiting for the page to be released rather than actually performing the swap.

Implementing and evaluating other existing DSM solutions, such as the Home Based or Homeless Lazy Release consistency protocol[54], in the context of the DVM is an interesting future research topic. Another topic which requires investigation is how tuning the existing implementation affects performance, such as varying the minimum page lease time. Preliminary testing shows that the effect of increasing the lease time is neglectable, since the majority of the possession time comes from prolonging an already expired lease, but this is not a fully explored topic in this thesis. Using a value of 20μs or less tends to have a negative effect on performance.

5.5 Improving the IBPP framework

Currently, the IBPP framework performs at an acceptable level for the operations utilized by the DVM. It is not, however, fast enough in general to be a competitive library for use in general RDMA applications. It performs worse than both the performance test package supplied with the OFED stack, as well as other IB-based implementations, such as MVAPICH2.

As previously stated, there is a lot of optimization work remaining on the overall library, as well as features that need to be supported for it to be generally useable. The first major problem is the high overhead for send and receive operations. This is the main bottleneck of the library, and definitely has to be addressed before releasing the library to the public.

Secondly, the framework currently only supports reliable connections. The library will have to be extended to support the other connection modes listed in table 2.2. Most importantly, UD connections must be available to utilize the multi cast capabilities of IB.

5.6 Writing a compiler framework for the DVM

The DASS is designed to be a target for compilation. In order to make developing applications for the DVM feasible, it is obvious that a solid compiler framework will have to be established for the system. The easiest way to do this would be to write a modification to the LLVM[37] target independent code generator. The work load of this task is not too great, and the LLVM already provides optimalization and support for a wide array of higher level languages. DVM-specific calls and functionality, such as the synchronization primitives, could be exposed to the language through use of library calls with inline ASM, for instance. When making the modifications to LLVM, a lot of DVM-specific optimalization could be implemented.

Another route to take is to design a new high-level language, with the specific DVM characteristics in mind from the beginning of the design process. This is quite a large process, and might very well be redundant, since compiling existing languages into DASS should be unproblematic. But it leaves more room for language-level support for the DVM synchronization tools, the architecture and other properties regarding the DSM nature of the DVM.

5.7 Application control

The current implementation of the DVM has little application control built in. Each node will load the program specified in the topology file, and start execution as soon as all nodes are connected.

For deployment of large scale applications, a set of control messages will

have to be implemented. A monitoring system, where information about the current load of the nodes in the system is available, is needed to know where to spawn threads of execution. Other runtime events, such as errors, exceptions and swap-out messages to notify nodes that they no longer are responsible for storing a memory page, has to be implemented. This can be achieved by either having the new owner send a swap-out message to the old node, or by notifying the pagenode of the swap through an immediate value sent with the write operation updating the page table. The pagenode could then take responsibility of notifying the previous owner.

One interesting future development is to allow multiple applications to run inside a single virtual machine instance. This can be accomplished by splitting the address format into an additional field, where the topmost bits represent an application ID. This would add a slight overhead in address lookup, as verification of ownership would have to be confirmed before allowing access to the memory.

Allowing multiple applications inside a VM would enable the DVM to interface with existing job schedulers such as the Oracle Grid Engine [8].

5.8 Local resource sharing

Currently, the DVM has only two instructions utilizing local resources. The *out* and *in* operations will print to the local console, and query input from the local input device respectively. For real world applications, support for sharing resources such as file handles, network sockets and other peripherals will have to be made available to the DVM.

For most resources, such as file handles and sockets, implementing a mapping scheme similar to the memory mapping model would be possible. Access locks can protect the resources from simultaneous access, and writing with immediate notifications to a designated array of buffers will allow fairly fast output to the resources. Similarly, a buffer array can be designated to hold the data received from, say, a local socket or a buffered file read, allowing the current owner to access the data using RDMA reads.

Input resources, such as mouse position, keyboard states and other peripherals can be mapped directly to a designated memory region, allowing all nodes to poll their state using singular read operations.

Other shared resources can be created by the DVM itself, such as video buffers or other buffers. Using a video buffer to hold bitmaps that are swapped to the local display can for instance allow rendering of graphics to be delegated to different nodes than the front-end node rendering the image.

5.9 Threading

The distributed assembly language is designed to be usable as a compilation-target for higher level languages. It supports most basic x86 style operations, but it also provides additional features such as multi-register processing and built in threading functionality.

Providing assembly-level synchronization mechanisms is important due to the distributed nature of the runtime environment. While memory access

on the shared memory is atomic by nature, it is subject to race conditions. Additionally, using variables on the heap as tools for synchronization is expensive because of the high latency of accessing non-local memory pages. By using InfiniBand atomic operations and the IBPP framework to expose atomic operations to the bytecode, we provide compilers with a powerful set of tools when generating DASS from higher level languages.

As of now, the synchronization instructions and underlying API has not been implemented. It requires some additional globally accessible storage to be provided, but this could easily be delegated to the paging nodes. For instance, by letting one pagenode take responsibility for making an RDMA-accessible buffer of 64 bit integers available, it could be used for storing barriers, mutexes and semaphores. Implementing access to these variables would be fairly trivial in the DVM, as the IBPP framework offers most of the functionality required.

5.10 Going big - large scale applications and clusters

The testing that has been done in this thesis was conducted on a relatively small InfiniBand cluster. As the DVM is designed to utilize the processing and memory capabilities provided by large scale HPC clusters, we need to take a look at how scaling affects the performance.

The processing nodes in the virtual machine are fully interconnected, and the actual data transfer of memory pages is performed on the direct links between them. The load on each individual link is limited by the swapping

rate of the memory pages. Given that the cost of a page swap is roughly $20\mu s$, combined with the page lease time, the network load should not become an issue.

The page nodes will have less load in terms of consumed bandwidth, but the number of requests per pagenode could become very high for popular regions of memory. As all nodes requesting access need to perform *at least* one atomic and one read operation per access, this could become a problem for the most active memory sections.

The simple solution to the problem of overloading pagenodes is to perform runtime profiling of the memory load, and adjust the number of memory pages the most heavily loaded nodes have responsibility for. The adjustment could possibly be done during runtime, and if not, the application configuration could take these issues into account, adjusting the designations in the node topology file for future executions of the application. Programmatically adjusting the number of page nodes versus compute nodes could be a possible improvement.

Being a fully interconnected network of nodes, the DVM would ideally run on a fully interconnected cluster. This is, however, not feasible for large scale HPC clusters, as the cost would be incredibly high. We propose using a topology such as the Dragonfly[36], which provides a low diameter highly scalable network, well suited for the DVM.

The Dragonfly topology, which we can see in figure 5.1, divides the network into $N=a*(k-(a-1))/2$ virtual routers, each of which consist of a radix k routers. Every virtual router has $N-1$ broad links, one for every other virtual router in the network. The internal topology of each router varies, but the most promising approach is to use a fully interconnected internal topology,

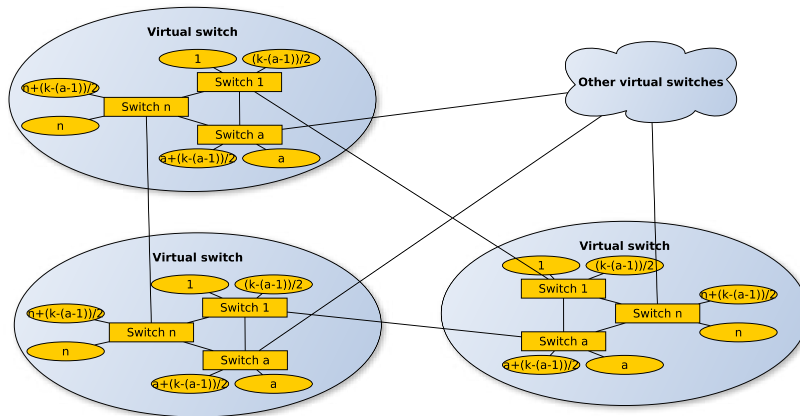

Figure 5.1: The Dragonfly topology

where each router in the group has *a* links to other routers, *(k - (a-1))/2* end nodes and *(k-(a-1))/2* links to other virtual routers.

The low hop count, combined with the high radix of the virtual routers, makes the dragonfly highly suitable for applications such as the DVM, with low bandwidth high frequency communication between random pairs of nodes.

5.11 Some final thoughts

During the development phase of this project, a lot of ideas has been conceived, weighted and discarded. For instance, I originally intended to utilize the hardware supported multicast of IB to propagate changes in page ownership, but fell back to the pagenode solution used in the current implementation.

I also planned to use the shared memory of the DVM for the process stack

as well, but during testing it proved to be much too slow for what the stack is traditionally used for, such as temporary storage of local loop variables. This caused me to settle for a local stack, with a much lower access time than the shared memory operations. This results in a limitation on accessing stack variables, namely that data stored on the stack is local only, and lies outside the applications virtual shared memory.

Chapter 6

Conclusion

In this thesis, we have developed and tested both an IB programming framework and library, and a DVM utilizing this library. We have discussed the various properties of the developed systems, and explained their design and the philosophy behind them.

As we discovered, it is possible, and even feasible, to use a DSM to create a VM with a SSI spanning multiple nodes in a cluster with an IB interconnection network. This is achieved despite the lack of fully optimized subsystems, proving that given time and further development, the DVM can become a viable development platform for large scale systems that would normally require advanced distributed programming schemes.

We have also proposed and discussed an assembly language to be used as a compilation target for higher level languages, with support for synchronization primitives and other threading and distributed operations.

6.1 IBPP contributions and future work

During the work on this thesis, we have designed and created the IBPP framework, providing simplified access to IB verb operations in C++. The IBPP framework is a promising start, and it should be possible to develop it into a viable open source library. This may open possibilities for developers to take IB technology in use for a larger variety of applications. As previously mentioned in this thesis, the IBPP framework is in need of optimization, but making it a viable library in the future is not unrealistic.

6.2 InfiniBand and RDMA as a platform for DSM systems

As documented in literature [48] and from what we have observed during this thesis, the InfiniBand RDMA operations and other VIA-based networking platforms are well suited for applications such as the DVM, and specifically for DSM systems where low latency remote memory access is essential. Additionally, the lack of kernel interruption for one-sided communication is very well suited for these applications.

In chapter 4 we saw that the memory access speed of the DVM is around 5 times slower than regular C *memcpy* operations. The access time for memory currently possessed by another node is still quite high, around 1500μs, and this will have to be addressed in the future. Proper synchronization mechanisms should alleviate this, though.

6.3 The future of the DVM and closing words

The DVM developed in this thesis is academic in nature, and does not aim to compete with existing commercial solutions such as the vSMP from ScaleMP[50]. It does, however, provide an open platform for future research into many topics, such as the underlying DSM, how to efficiently design and implement a proper page-based distributed runtime system and making compilers capable of masking the distributed properties from the programmer by compiling to the DASS platform.

I do not know, and will not guess at the future of the DVM or the IBPP framework, but it is my intention at the very least to release IBPP as an open source library, albeit in a slightly more streamlined and optimized version than the current implementation.

Appendix A

The DVM instruction set

Here is an overview of the instruction set supported by the DVM. Instructions marked with a * are not currently implemented.

Instruction	Operands	Description
set	d s	Sets the value of d or [d] to the value of s, the register s or the array-index [s].
in	d	Queries for console input and stores the integer value in d.
out	s	Print the value of s to the local console.
clk	d	Store the number of microseconds since the thread started executing into s or [s].

Table A.1: Register instructions

Instruction	Operands	Description
lod	s d n	Loads the values starting at s to s + (n-1) * type size into registers d to d + n - 1
pop	d n	Pops a range of registers from the top of the stack into register d to d + n - 1. Pop inverses the order of push, so after pushing and popping n registers, all registers will retain their value.
push	s n	Pushes all the registers from s to s + n - 1 onto the local stack.
str	d s n	Stores n registers from s to s + n - 1 to the memory addresses d to d+ (n-1) * type size.

Table A.2: Memory instructions

Instruction	Operands	Description
and	d s	store the value of d and s into d.
or	d s	Store the value of d or s into d.
xor	d s	Store the value of d xor s into d.
shl	d s	Shift d left by s bits.
shr	d s	Shift d right by s bits

Table A.3: Bitwise instructions

Instruction	Operands	Description
add	d s	Adds the value of, register s or [s] to d. All of the arithmetic instructions take either a value, source register or source register index as the source, and either a register or register index as destination.
sub	d s	Subtract s from d.
mul	d s	Multiply d with s.
div	d s	Divide d by s.
inc	d	Increase d by 1.
dec	d	Decrease d by 1.

Table A.4: Arithmetic instructions

Instruction	Operands	Description
cmp	s1 s2	Compares s1 and s2, and sets the appropriate flags.
jmp	l	Jumps to the first instruction after jump label l.
jlt	s1 s2 l	If s1 < s2, jump to l.
jgt	s1 s2 l	If s1 > s2, jump to l.
jeq	s1 s2 l	If s1 == s2, jump to l.
jne	s1 s2 l	If s1 != s2, jump to l.
jle	s1 s2 l	If s1 <= s2, jump to l.
jge	s1 s2 l	If s1 >= s2, jump to l.
cjl	s1 s2 l	Compare s1 and s2, and jumps to l if s1 < s2.
cjg	s1 s2 l	Compare s1 and s2, and jumps to l if s1 > s2.
cje	s1 s2 l	Compare s1 and s2, and jumps to l if s1 == s2.
for	s1 s2 l	While s1 is less than s2, increase s1 and jump to l.
hlt		Terminate program execution.

Table A.5: Control instructions

Instruction	Operands	Description
*frk**	l d	Start a new thread of execution at jump point l. The ID of the created thread is stored in d.
*join**	d	Wait for thread d to finish execution.
*thid**	d	Place the ID of the current thread into d
*barr**	d n	Wait at barrier d until n processes join it. Barrier creation is implicit, i.e the first process to wait on a barrier creates it.
*lock**	d	Acquire mutex d.
*unlock**	d	Unlock mutex d.
*wait**	d	Wait for a signal on d.
*signal**	d	Signal d.

Table A.6: Threading and Synchronization instructions

Bibliography

[1] Choosing between OpenMP* and Explicit Threading Methods. *http://software.intel.com/en-us/articles/choosing-between-openmp-and-explicit-threading-methods/*.

[2] History of the Java technology. *http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html*.

[3] HotspotVM. *http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html*.

[4] MVAPICH2. *http://mvapich.cse.ohio-state.edu/overview/mvapich2/*.

[5] OpenFabrics Alliance. *https://www.openfabrics.org*.

[6] OpenFabrics Enterprise Distribution. *https://www.openfabrics.org/resources/ofed-for-linux-ofed-for-windows/ofed-overview.html*.

[7] OpenMP. *http://www.openmp.org*.

[8] Oracle Grid Engine. *http://www.oracle.com/technetwork/oem/grid-engine-166852.html*.

[9] The boost libraries. *http://www.boost.org*.

[10] The Kaffe VM. *http://www.kaffe.org*.

[11] Top 500 list November 2011. *http://top500.org/list/2011/11/100*.

[12] VMWare. *http://www.vmware.com*.

[13] ADVE, S. V., AND GHARACHORLOO, K. Shared memory consistency models: A tutorial. *Computer 29* (December 1996), 66–76.

[14] ARIDOR, Y., FACTOR, M., AND TEPERMAN, A. cjvm: a single system image of a jvm on a cluster. In *In Proceedings of the International Conference on Parallel Processing* (1999), pp. 4–11.

[15] AYCOCK, J. A brief history of just-in-time. *ACM Comput. Surv. 35* (June 2003), 97–113.

[16] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev. 37* (Oct. 2003), 164–177.

[17] BARTHOLOMEW, D. Qemu: a multihost, multitarget emulator. *Linux J. 2006* (May 2006), 3–.

[18] BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHEK, R., AND SUNDERAM, V. A user"s guide to pvm parallel virtual machine. Tech. rep., Knoxville, TN, USA, 1991.

[19] BELL, J. R. Threaded code. *Commun. ACM 16* (June 1973), 370–372.

[20] BLACK, A. P., HUTCHINSON, N. C., JUL, E., AND LEVY, H. M. The development of the emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 2007), HOPL III, ACM, pp. 11–1–11–51.

[21] BOX, D., AND PATTISON, T. *Essential .NET: The Common Language Runtime*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[22] Cheung, B. W.-L., Wang, C.-L., and Lau", F. C.-M. "migrating-home protocol for software distributed shared memory.". *"J. Inf. Sci. Eng."* (2002), 929–957.

[23] David Fair, Intel, Manoj Wadekar, QLogic, and Blaine Kohl, Ethernet Alliance. iWARP Brings Low-Latency Fabric Technology to Ethernet, white paper. `http://www.ethernetalliance.org/wp-content/uploads/2011/10/iWARP-Low-Latency.pdf`.

[24] Davis, B., Beatty, A., Casey, K., Gregg, D., and Waldron, J. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators* (New York, NY, USA, 2003), IVME '03, ACM, pp. 41–49.

[25] Devulapalli, A. Distributed queue-based locking using advanced network features. In *Proceedings of the 2005 International Conference on Parallel Processing* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 408–415.

[26] Ertl, M. A., and Gregg, D. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *SIGPLAN Not. 38* (May 2003), 278–288.

[27] Factor, M., Schuster, A., and Shagin, K. Javasplit: A runtime for execution of monolithic java programs on heterogeneous collections of commodity workstations. *Cluster Computing, IEEE International Conference on 0* (2003), 110.

[28] Frey, P., and Alonso, G. Minimizing the hidden cost of rdma. In *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on* (june 2009), pp. 553 –560.

[29] Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R., Daniel, D., Graham, R., and Woodall,

T. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004, pp. 353–377. 10.1007/978-3-540-30218-6_19.

[30] Gharachorloo, K. The plight of software distributed shared memory. In *Workshop on Software Distributed Shared Memory* (1999).

[31] InfiniBand Trade Association. Infiniband(tm) architecture specification release 1.2.1. *http://www.infinibandta.org/content/pages.php?pg=technology_download*.

[32] Intel. Intel®64 and IA-32 Architectures Software Developer Manuals . *http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html*.

[33] Intel, Compaq, and Microsoft. Virtual Interface Architecture Specification 1.0, 1997.

[34] Jiang, W., Liu, J., Jin, H.-W., Panda, D. K., Buntinas, D., Thakur, R., and Gropp, W. D. Efficient implementation of mpi-2 passive one-sided communication on infiniband clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings* (2004), D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241 of *Lecture Notes in Computer Science*, Springer, pp. 68–76.

[35] Keleher, P. J. *Lazy release consistency for distributed shared memory*. PhD thesis, Houston, TX, USA, 1995. UMI Order No. GAX96-10659.

[36] Kim, J., Dally, W. J., Scott, S., and Abts, D. Technology-driven, highly-scalable dragonfly topology. *SIGARCH Comput. Archit. News 36* (June 2008), 77–88.

[37] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–.

[38] Li, K., and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7* (November 1989), 321–359.

[39] Liss, L., Birk, Y., and Schuster, A. Efficient exploitation of kernel access to infiniband: a software dsm example. In *In proceedings on High Performance Interconnects, 2003. 11th Symposium* (2003).

[40] Liu, J., Mamidala, A., Vishnu, A., and Panda, D. K. Performance evaluation of infiniband with pci express. *IEEE Micro 25* (2004), 2005.

[41] Maydan, D., Amarsinghe, S., and Lam, M. Data dependence and data-flow analysis of arrays. In *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., vol. 757 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1993, pp. 434–448.

[42] Mellanox. RDMA aware programming user manual. *http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf*.

[43] Mietke, F., Rex, R., Baumgartl, R., Mehlan, T., Hoefler, T., and Rehm, W. Analysis of the memory registration process in the mellanox infiniband software stack. In *Proceedings of the 12th international conference on Parallel Processing* (Berlin, Heidelberg, 2006), Euro-Par'06, Springer-Verlag, pp. 124–133.

[44] Noronha, R. M., and P, D. K. Reducing diff overhead in software dsm systems using rdma operations in infiniband. In *In Workshop*

on Remote Direct Memory Access (RDMA): RAIT 2004, (Cluster '04 (2004).

[45] Osendorfer, C., Tao, J., Trinitis, C., and Mairandres, M. Vismi: Software distributed shared memory for infiniband clusters. In *Proceedings of the Network Computing and Applications, Third IEEE International Symposium* (Washington, DC, USA, 2004), NCA '04, IEEE Computer Society, pp. 185–191.

[46] Park, I., and Wook Kim, S. Note: The distributed virtual shared-memory system based on the infiniband architecture. *J. Parallel Distrib. Comput. 65* (October 2005), 1271–1280.

[47] Proebsting, T. A. Optimizing an ansi c interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1995), POPL '95, ACM, pp. 322–332.

[48] Rangarajan, M., and Iftode, L. Software distributed shared memory over virtual interface architecture: Implementation and performance. In *IN PROCEEDINGS OF THE 3RD EXTREME LINUX WORKSHOP* (2000), pp. 341–352.

[49] Sato, M., Harada, H., Hasegawa, A., and Ishikawa, Y. Cluster-enabled openmp: An openmp compiler for the scash software distributed shared memory system. *Sci. Program. 9* (August 2001), 123–130.

[50] Schmidl, D., Terboven, C., Wolf, A., Mey, D. a., and Bischof, C. How to scale nested openmp applications on the scalemp vsmp architecture. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing* (Washington, DC, USA, 2010), CLUSTER '10, IEEE Computer Society, pp. 29–37.

[51] Snir, M. *MPI–the Complete Reference: The MPI-2 extentions.* No. v. 2 in Scientific and engineering computation. MIT Press, 1998.

[52] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. (revised) ed. MIT Press, Cambridge, MA, USA, 1998.

[53] Tao, J., Karl, W., and Trinitis, C. Implementing an openmp execution environment on infiniband clusters. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming* (Berlin, Heidelberg, 2008), IWOMP'05/IWOMP'06, Springer-Verlag, pp. 65–77.

[54] Yu, B.-H., Huang, Z., Cranefield, S., and Purvis, M. Homeless and home-based lazy release consistency protocols on distributed shared memory. In *Proceedings of the 27th Australasian conference on Computer science - Volume 26* (Darlinghurst, Australia, Australia, 2004), ACSC '04, Australian Computer Society, Inc., pp. 117–123.

[55] Yu, W., and Cox, A. Java/dsm: A platform for heterogeneous computing. *Concurrency: Practice and Experience 9*, 11 (1997), 1213–1224.

[56] Yun, H.-C., Lee, S.-K., Lee, J., and Maeng, S. An efficient lock protocol for home-based lazy release consistency. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid* (Washington, DC, USA, 2001), CCGRID '01, IEEE Computer Society, pp. 527–.

[57] Zhu, W., Wang, C.-L., and Lau, F. C. M. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Proceedings of the IEEE International Conference on Cluster Computing* (Washington, DC, USA, 2002), CLUSTER '02, IEEE Computer Society, pp. 381–.