An Efficient Open MPI Transport System for Virtual Worker Nodes

Timothy Hayes
B.A. (Mod.) Computer Science
Trinity College Dublin
Final Year Project, April 2009
Supervisor: Dr. Stephen Childs

"It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years."

– *John Von Neumann, circa 1949*

Acknowledgements

I would like to thank my supervisor Dr. Stephen Childs for agreeing to supervise this project and for all his helpful advice throughout the year. I would also like to thank the people of the Open MPI and Xen developer mailing lists; without their time and enthusiasm this project would surely not have seen completion. Finally I would like to thank my family and friends for their support and encouragement over the past year.

Abstract

There is increased interest in running cluster jobs within virtual machines to provide isolation and customised environments. For this to be successful, it is essential that efficient transport mechanisms are available. In particular, communication between virtual machines on the same physical host should provide similar performance to that between multiple jobs running on a single host. This project will implement a shared memory transport for Open MPI that uses Xen features to provide low-latency communication between virtual machines. Open MPI's failover features will be used to automatically switch to other transport mechanisms (e.g. Ethernet) when virtual machines are migrated so they no longer share a physical host.

Contents

List of Figures

List of Tables

1 Introduction

There has been an interest in virtualizing worker nodes in high performance computing (HPC) clusters for a variety of reasons. Virtual machines can provide different customised environments sharing the same hardware. They can easily accommodate fine-tuning an operating system's scheduling and memory management for specific applications. Many light-weight, customised guest operating systems can be used to partition large multi-core machines. Virtual machines also enable checkpointing and migration of a job's entire environment; this can be used to implement transparent real-time load balancing.

While there are many benefits to fusing virtualization and high-performance computing, there are also reasons not to. Applications running on a virtual machine incur performance overheads not present on a native operating system. These issues are being looked at in the form of hardware solutions (Intel, AMD and IBM now have virtualization features in their processors) along with extensive research into paravirtualized software (e.g. Xen and the IBM Research Hypervisor)[7]. One particular performance issue is inter-VM (i.e. virtual machines on the same physical machine) communication; this theoretically could perform as well as inter-process communication, yet remains highly inefficient.

Performance overhead is probably the greatest factor inhibiting the HPC community from adopting virtualized solutions. HPC applications rely heavily on efficient communication channels between processes. The desire is to have low latency and high throughput point-to-point links between processes which can be guaranteed by inter-process communication but not inter-VM communication. The motivation of this project is to try to remove some of the overhead that exists in virtualized clusters while making the optimisation completely transparent to existing code. This would provide an incentive to the HPC community to adopt virtualized solutions.

Message Passing Interface (MPI) is a standardised and widely adopted communication interface used by many HPC clusters. It has been used to develop solutions for huge problems such as climate change prediction and simulating drug compound effects on malaria. This project is concerned with running MPI programs on virtualized environments hosted by Xen. Although Xen offers near-native performance to its virtual machines, communication channels between virtual machines on the same physical machine are still prone to inefficiencies. Using an MPI implementation called Open MPI, this project implemented a highly efficient MPI communication system specifically for Xen virtual machines. This came in the form of an easily installable plug-in which outperforms the native transfer mechanism in terms of latency, throughput and CPU utilisation.

This report can be read by itself or as a complement to the source code. Section 2 describes the primary technologies used in the scope of this project. Section 3

analyses the current inter-VM communication system in Xen. It identifies critical weaknesses in the basic implementation and describes third-party software that addresses them. Section 4 describes a subset of the Open MPI and Xen architectures that are crucial to the project's implementation. Section 5 draws on material from section 4 and describes the design process of the plug-in itself. Section 6 provides a detailed analysis of the plug-in's performance. Section 7 discusses some of the pitfalls encountered during the design and implementation of the project; finally, a summary of the project as a whole is given.

2 Background Information

This section introduces technologies and terminology of significance to the project.

2.1 Message Passing Interface

Message Passing Interface (MPI) is an API to aid the building of parallel and distributed programs. The model allows individual processes to work together to help solve a common task or problem; processes communicate with each other by passing messages. In contrast to shared-memory models of parallel programming, MPI processes neednt be on the same physical machine. For example, in a threaded program every thread must exist in the same physical memory whereas with MPI, the processes can be local or on remote machines. MPI is therefore more scalable and remains the de facto choice for programming high performance computers and clusters. MPI has no targeted language, although common bindings include C, C++ and Fortan. MPI has two standard versions defined by the MPI Forum, MPI-1[15] (currently MPI-1.2) and MPI-2[16] (currently MPI-2.1).

2.2 Open MPI

Open MPI[4, 5] is an implementation of both versions of the MPI API. It was a merger between four major MPI implementations and aspired to take the best features of each one into a single cutting-edge library. It is a widely adopted multi-platform MPI implementation that strives to provide high performance, high bandwidth and low latency on every supported system. It is an open source project with many contributors coming from both academic institutions and industry, however the Open MPI community encourage contributions from all people interested. Open MPI has a component based architecture so specific features and tasks are decoupled from each other and implemented cohesively with strict APIs, this way a programmer can develop their own features in the form of plug-ins.

2.3 Virtualization

Virtualization is a concept that refers to the abstraction of a physical machine's resources into multiple isolated copies known as virtual machines. A virtual machine can present a very similar (if not identical) hardware model as the physical machine it is running on. This gives the operating system the illusion of running directly on hardware. The benefit of this is that many virtual machines, each with their own operating system, can run concurrently on one physical machine.

A layer of code is inserted between the virtual machines and the underlying system which is known as a virtual machine monitor or hypervisor. There are two primary types of hypervisor: 'bare-metal' which runs on hardware directly in place of an operating system or 'hosted' which runs as a user application in an operating system.

Paravirtualization is a specific kind of virtualization where the exact hardware model can't be cloned verbatim, so instead a very similar one is used. For an operating system to run in this kind of virtual machine it must be modified so that all problematic machine instructions are replaced with a mixture of safe machine instructions and calls to the hypervisor itself. This way the operating system becomes aware that it is running on a virtual machine.

2.4 Xen

Xen[8] is bare-metal virtual machine monitor that supports a range of guest operating systems and CPU architectures; it was developed at University of Cambridge and publicly made first available in 2003. Xen offers a choice to its end users in that it can run paravirtualized virtual machines for operating systems that have been made Xen aware (this particularly common for Linux based operating systems which has an open source kernel implementation) and also hardware assisted virtual machines for operating systems which can't be modified e.g. Windows.

The Xen hypervisor is a very thin layer that sits between the hardware and the virtual machines. Interacting with the hypervisor requires the presence of a specific targeted virtual machine with a paravirtualized operating system. In Xen terminology this is known as the dom0 (short for domain 0) and every other virtual machine is then called a domU (for unprivileged domain). The dom0 can be a variety of modified operating systems including OpenSolaris and NetBSD, however Linux based systems remain the most common choice. The dom0 is the first virtual machine booted by Xen and includes applications to create, delete and manage virtual machines.

Xen offers its virtual machines various hardware resources with a split device driver model. What this means is that a guest operating system can interact with their device drivers as normal, however instead of the device drivers interacting directly with the hardware, the requests will be passed via the hypervisor to a special virtual machine known as the driver domain. The driver domain contains all the real device drivers that interact with the hardware and also logic to help multiplex requests from the unprivileged domains in order to share the physical resources of the machine. This way, all the virtual machines are given the illusion of interacting directly with real devices when in reality all their requests are tunnelled through a special domain first. More often than not, the dom0 is also the driver domain, however they can be separated which can help make the dom0 more

stable[2, p.47-74].

Xen also offers a facility known as live migration where a virtual machine running on one hypervisor can change physical hosts and move to another hypervisor. The migration happens in real-time and is completely transparent to the virtual machine itself. The benefit of this is that virtual machines can be scheduled on different machines dynamically as a form of load balancing. For example, one could have a large number of virtual machines hosted on one hypervisor doing no important work; should one machine become active and require more resources, it can be migrated to another physical machine that can guarantee these resources.

3 Xen Inter-VM Communication

This section looks at how the Xen Inter-VM communication system works and summarises various third-party utilities that have attempted to improve it.

3.1 Xen - An Unenlightened Network

One of the purposes of a virtual machine monitor is to multiplex and time-share physical hardware to many virtual machines. Analogous to how an operating system provides the illusion to its processes that each one has complete and undisturbed access to physical hardware, a virtual machine monitor does the same for its virtual machines. For virtual machines running in Xen, one of the more interesting aspects is how it addressed the issue of per-guest networking with a limited amount of physical network interface cards available. For simplicity we will assume these to be Ethernet cards, however this could apply to any number of different technologies. A virtual machine should be isolated and ideally have no knowledge that it is sharing resources, therefore it is inevitable that each guest needs its own network device to access a local network.

One solution would be to have a separate network interface card for every virtual machine running, but naturally this solution would be expensive and would not scale well. Another solution would be to give each virtual machine the same device driver to access the network card directly. This would ultimately lead to disaster as every virtual machine would be competing to send and receive frames in unison leading to race conditions and fatal consequences. Fortunately Xen offers a clean and effective way for each virtual machine to exist on the same network using a single network interface card whilst giving each one the illusion of having their own network interface card.

Virtual machine guests on Xen talk directly to a virtual network device. This device essentially comes in the form of a device driver that offers the same functionality found in most generic Ethernet card drivers. Instead of translating instructions into hardware signals, the driver will interact with the Xen hypervisor in order to communicate with a corresponding back-end interface that exists in the driver domain. The back - interface translates instructions into regular MAC Ethernet frames and alerts a virtual software bridge, also running in the driver domain, of the request. The virtual bridge operates like any layer 2 switch; if it sees a frame is destined for a machine on a local network it will pass the frame to a real device driver corresponding to a physical network device. Each virtual network interface has a unique MAC address (usually randomly generated) and the whole process runs transparently to any user applications running in the virtual machines. See figure 3.1.

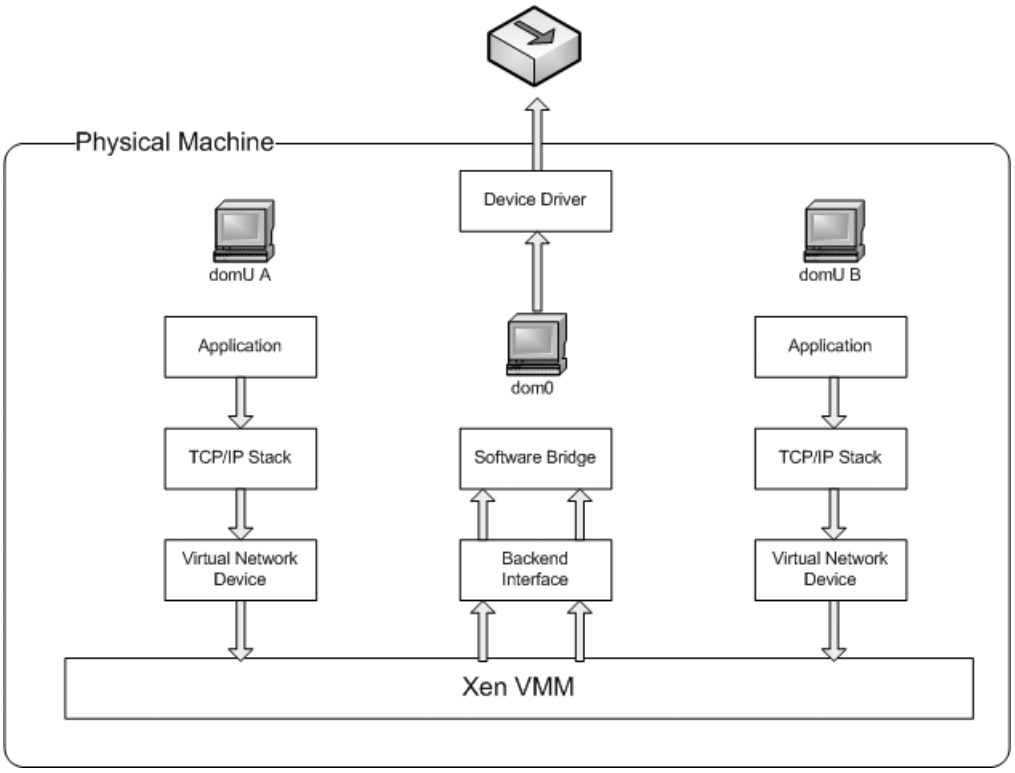

Figure 3.1: Process of Xen virtual machines accessing a local network

The main concern of this project is how virtual machines sitting in the same physical machine communicate with each other. The communication channels between a virtual machine to the outside network may be an acceptable solution since the extra complexity is somewhat lost in the general overhead of network bound traffic. On the other hand, it proves to be a very ineffective method for communication channels between virtual machines residing on the same physical host. Because these virtual machines share the same physical memory, one would expect performance close to inter-process communication. Figure 3.2 illustrates the problem. The main point here is that domU A doesn't know domU B is a virtual machine hosted on the same physical machine. Therefore domU A sends a packet of information to domU B under the assumption it exists somewhere on the LAN which leads to many inefficiencies[9, 10].

1. An application in domU A must make a system call and change into kernel space.

2. The kernel copies the data from the user's memory space into the kernel's memory space.

3. The data is then packed with TCP and then IP metadata before being passed to the virtual device driver.

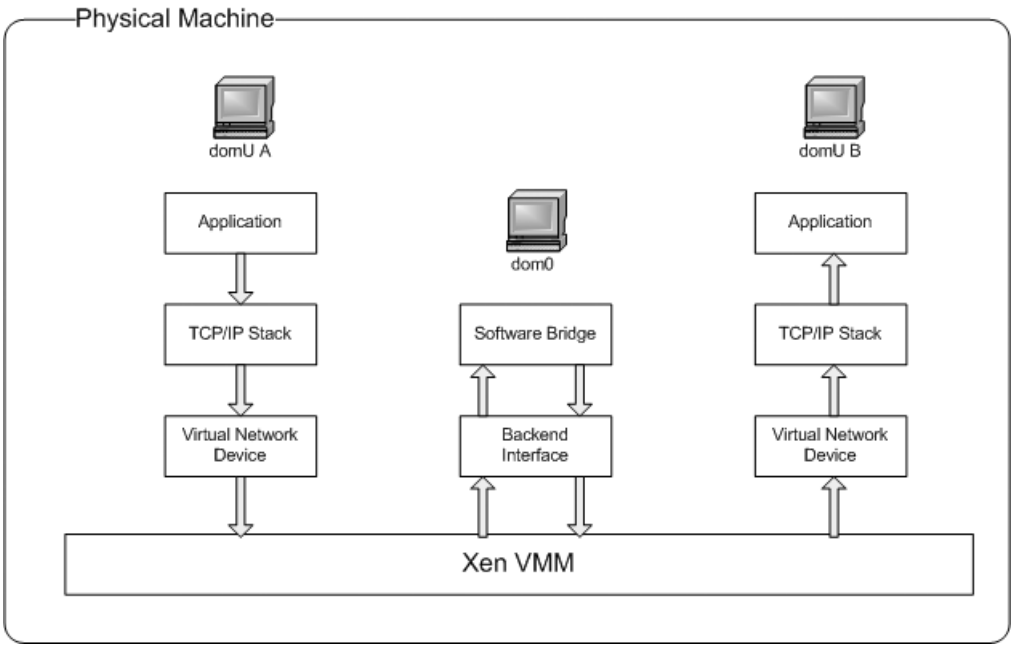

Figure 3.2: Process of Xen virtual machines communicating with each other

4. The virtual device driver surrounds the packet in an Ethernet frame.
5. The virtual device driver must copy this data into numerous free pages and then request that Xen map the page ownership from domU A to dom0. This is known as page flipping.
6. dom0 is then signalled and scheduled to handle this request.
7. The frame moves from the back-end driver to the software bridge.
8. The software bridge sees the destination resides in a virtual machine on the same physical machine so calls the back-end driver again.
9. The back-end driver places the frame into numerous free pages and requests Xen to map the page ownership from dom0 to domU B.
10. domU B is then signalled and scheduled to handle this request.
11. The frame is stripped of its Ethernet information and passed up into the TCP/IP stack.
12. The TCP/IP stack strips all the metadata from the packet.
13. The user-space application then makes a system call into kernel-space.
14. The kernel must copy the data from the kernel's memory space to the user's memory space.

All this is very long winded and inefficient way to essentially have two processes share a piece of information. It results in low bandwidth, high latency and far too much CPU utilisation. In the context of high performance computing the goal is to achieve the exact opposite: high bandwidth, low latency and as little CPU utilisation as possible. This project is concerned with bypassing the standard internal communication model of Xen for jobs running in Open MPI.

3.2 Previous Work

The following describes three third-party utilities that were developed to try and remedy the problems of Xen's inter-VM communication system.

3.2.1 XenLoop

XenLoop[12] is a utility developed at Binghamton University, NY. It modifies the network stack of the domU clients and inserts a module between the network layer and the data-link layer. This module contains a table with information on co-resident virtual machines. In order to fill this table with relevant information, when a guest is created or migrated onto a machine, they advertise their desire to participate in XenLoop using the XenStore database. A daemon runs in dom0 looking for changes in XenStore. If a new XenLoop enabled guest is detected or removed, the dom0 signals all resident guest domains which promptly create or destroy direct channels with the guest.

Advantages

- No page flipping.
- Works transparently for all existing applications.
- Handles live migration (in/out).
- Source code available.

Disadvantages

- Still goes through TCP/IP stack (unnecessary overhead & fragmentation).
- Relies heavily on the dom0 daemon.
- No granularity of control.

3.2.2 Xway

Xway[10, 11] is a self-contained kernel module for a guest domain developed by the ETRI, Republic of Korea. Its goal is to provide efficient inter-domain communication transparently to the guest operating systems. The basic idea is that an extra layer is inserted between the INET layer and TCP/IP stack. The layer redirects the flow of information down its own Xway protocol for traffic bound for inter-VM domains and alternatively redirects down the TCP/IP stack for traffic bound for machines external to the hypervisor. Inside the Xway stack, memory and event channels are mapped between guests using an out of band TCP channel.

Advantages

- No page flipping.
- Bi-directional socket interface.
- Bypasses the TCP/IP stack.
- Binary compatibility with the socket interface.
- Source code available.

Disadvantages

- Uses an out-of-band TCP channel (this is an extra overhead in the context of this project as Open MPI already has an efficient out-of-band channel system).
- Requires kernel modification.
- Relatively complicated design.
- Source code unmaintained and obsolete.
- Live migration was proposed for version 0.70 targeted for release in March 2008 but this was never released publicly.

3.2.3 XenSocket

XenSocket[13] is a loadable kernel module, developed at the IBM T.J Watson Research Center, NY. The module is a custom Berkeley socket implementation that creates direct paths between co-resident virtual machines. Sockets are unidirectional and therefore two are required for full-duplex communication. The receiving socket first allocates a pool of memory and grants access to it to another virtual machine. The binding function returns a unique identifier called a grant reference. The sending socket, created by the other virtual machine, maps the memory into its own address space using the grant reference. The socket avoids `AF_INET` (and thus TCP/IP stack) entirely instead using its own socket family `AF_XEN`. Because the sockets must be bound to particular domains, some sort of out-of-band system

is needed to transfer addressing information. The shared pool of memory is used as a circular buffer. The send and receive calls are blocking only.

Advantages

- No page flipping.
- Completely separated from the TCP/IP stack.
- No dependency on dom0.
- No out-of-band channels (more flexibility this way).
- Simple but efficient design.
- Source code available.

Disadvantages

- Unidirectional stream.
- Blocking send/receive system only.
- No notification of data.
- Applications need modification.
- Needs some means of communicating domids and grant references.
- Socket breaks if virtual machine emigrates from its original hypervisor.

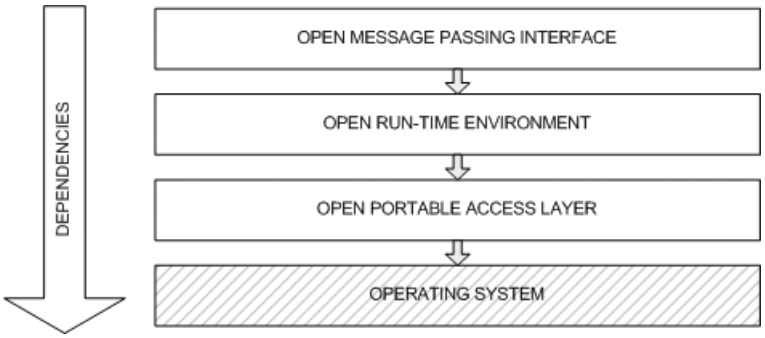

Figure 4.1: Open MPI code sections

4 Design Research

This section provides a study on subsets of the Open MPI and Xen architectures that are specifically related to this project.

4.1 Introduction to the Open MPI architecture

Open MPI is written with a component architecture in mind. From a high level analysis the library has 3 separate code sections. These are not layers in the purest sense of the word, but rather dependencies. See figure 4.1.

- Open MPI (OMPI) - The uppermost layer that contains the actual implementation of the MPI API.
- Open Run-time Environment (ORTE) - The middle layer which is a common interface to the runtime system. It is responsible for managing tasks such as process launching/discovery, out of band signalling, resource management.
- Open Portability Access Layer (OPAL) - The bottom layer which contains mostly utility code, e.g. a common C-style object management system, hash tables, linked lists, high resolution timers.

One of Open MPI's greatest strengths is its Modular Component Architecture (MCA). The MCA provides a light weight component architecture for the system and allows plug-ins with specific functionality to be found, loaded and unloaded dynamically. In Open MPI terminology a plug-in is generally known as a component.

A component has two incarnations: the component and the module. The component itself presents functions to open and close itself as well as an initialisa-

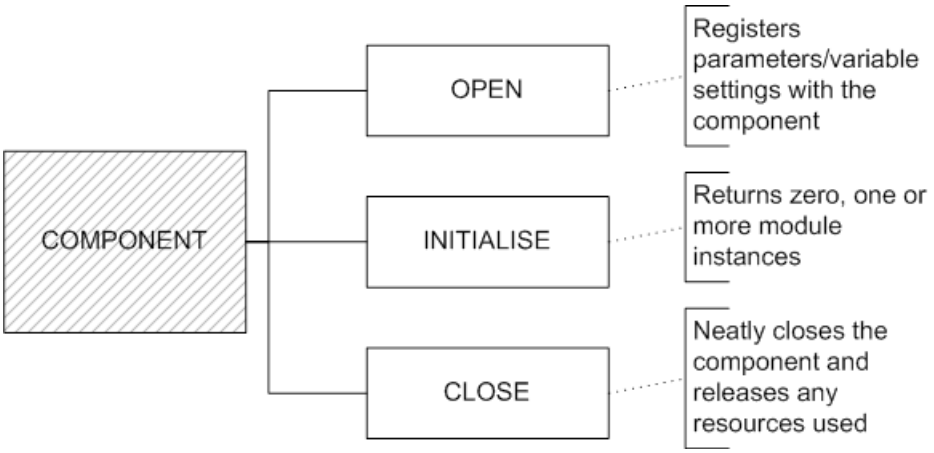

Figure 4.2: Functions available in an MCA component

tion function that is responsible for dynamically checking the suitability of being used, see figure 4.2. For example, there is a component to support point-to-point communication using InfiniBand technology, it is therefore necessary for the initialisation function to check the interface list for InfiniBand hardware. Modules on the other hand are the instances of the components which the rest of the system interact with. Where a component may be compared to a class, a module could be compared to an object. The component initialisation function can return zero or more modules. Returning no modules would suggest that there isn't an environment suitable to use the component. The component can optionally return multiple modules too; for example, if there are multiple network interface cards present, the component may wish to have one module managing each one individually.

Every module uses some public API specific to a particular framework and each framework is defined to have a targeted purpose. For example, there is a framework called *allocator* responsible for the allocation of memory. There are two components defined in this framework: basic - a simple heap based allocator and bucket - a bucket based allocator. Although internally they are logically different, they present the exact same public interface to the rest of the system. Some frameworks have only one component, for example the *Byte Management Layer (BML)* framework has one component called r2. r2 is used to open and multiplex all the components of the *Byte Transfer Layer (BTL)* framework. Having the framework and API defined like this means if one wants to change the internal logic for this particular task, they won't need to modify the core of the system.

This project is specifically focused on the *PML*, *BML* and *BTL* frameworks but primarily the *BTL* framework. There will be a brief overview of the functionality and uses of the *PML* and *BML* frameworks and then a more detailed look at the *BTL* framework.

4.1.1 The Point to Point Messaging Layer

The *Point to Point Messaging Layer (PML)* is a bridge between the MPI semantics (e.g. `MPI_Send()`) and the underlying transfer techniques. It is a relatively thin layer responsible for the scheduling and fragmentation of data. The MCA framework only selects one *PML* component which is then used for the duration of the job. Currently there are two main *PML* components written. `ob1` was written to be a progress engine and track multiple transport interfaces; for example, it can schedule and track messages destined for processes reachable via shared memory, TCP and any number of specialised interfaces (e.g. InfiniBand). `ob1` is specifically targeted to use *BTL* modules. The alternative is the `cm` component which doesn't use *BTL* modules at all, instead it uses a single *Matching Transport Layer (MTL)* module. An *MTL* component differs from a *BTL* component in that it is written for devices that natively support hardware/library message matching. The `cm` component won't provide features such as fragmentation, multi-device support and NIC failover, instead these features will be delegated to the *MTL* component. The `c1` component and *MTL* framework are out of the scope of this project and instead the focus will be on `ob1` component and the *BTL* framework.

A standard *PML* module has public interface outlined in table 4.1. This is achieved by using a structure containing pointers to functions. The *PML* functions have a strong resemblance to the `MPI_Send()`, `MPI_Recv()`, `MPI_Test()` and `MPI_Wait()` MPI functions.

The `ob1` module is internally a buffered and asynchronous transport engine. Although it offers both blocking and non-blocking send and receive functions to the rest of the Open MPI framework, internally it works by preregistering completion functions to be called by lower layers and then waiting/progressing until these callback functions are eventually invoked. For example, if the user's program called `MPI_ISend()`, `ob1` would simply invoke an asynchronous send with a *BTL* module and the user's program would then use `MPI_Wait()` or `MPI_Test()` in order to check for the invocation of callback function. On the other hand, if the user were to call `MPI_Send()`, `ob1` would still call an asynchronous send with a *BTL* module, however it won't return from the function until it detects the *BTL* module has invoked the callback function. Internally this works with condition variables, however instead of sleeping, `ob1` will attempt to progress and try and do some useful work each time before checking the condition.

Function	Description
pml_add_procs	Called to notify when new processes have been created
pml_del_procs	Called to notify when processes have been terminated
pml_enable	Called to enable the *PML* and *BTL* modules
pml_progress	Called to progress all the *BTL* modules used
pml_add_comm	Called to notify when a new communicator has been created
pml_del_comm	Called to notify when a communicator has been destroyed
pml_irecv_init	Create a non-blocking request to receive an MPI message without executing it
pml_isend_init	Create a non-blocking request to send an MPI message without executing it
pml_start	Execute multiple requests created by the `pml_irecv_init()` and `pml_isend_init()` functions
pml_irecv	Post a non-blocking request to receive an MPI message
pml_recv	A blocking request to receive an MPI message
pml_isend	Post a non-blocking request to send an MPI message
pml_send	A blocking request to send an MPI message
pml_iprobe	Perform a non-blocking poll for the completion of a receive
pml_probe	perform a blocking poll for the completion of a receive
pml_dump	Diagnostic information

Table 4.1: Functions available in a *PML* module

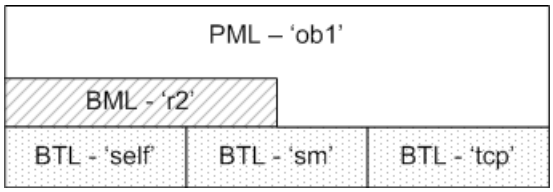

Figure 4.3: The hierarchy of the *PML*, *BML* & *BTL* frameworks

4.1.2 The BTL Management Layer

Although the *PML* is a very lightweight and simple framework, ob1 uses an additional module to help manage the *BTL* modules it utilises. The *BTL Management Layer (BML)* framework is in charge of opening and multiplexing multiple *BTL* modules. It schedules multiple *BTL* modules in a round-robin fashion to achieve a kind of load balancing. There is only one *BML* component currently written called r2. When ob1 has its functions `pml_add_procs()`, `pml_del_procs()` and `pml_progress()` called, it actually passes these requests down to r2. r2 will open all the *BTL* components during its initialisation phase and cache all the available *BTL* modules which are returned. This is particularly useful as *BTL* modules have an associated priority value, so r2 can find the best module for sending the message. It is also useful if there are multiple *BTL* modules returned from the same component (e.g. if a machine has two Ethernet cards, there would be a unique *BTL* module per card), r2 will alternate using modules of the same priority in order to distribute the load. Not every process may be reachable by a given *BTL* module (e.g. a shared memory module can only communicate with processes on the same host), r2 will query each *BTL* module about this and use the information in its scheduler. All this is made semi-transparent to ob1. See figure 4.3.

4.1.3 The Byte Transfer Layer

The *Byte Transfer Layer (BTL)* is a framework targeted to move raw data to and from processes. A *BTL* component is self-contained and should be completely unaware of any other components in the *BTL* framework. A *BTL* module is designed to utilise some specific transportation technique to move data and therefore a *BTL* module may only communicate with another *BTL* module of the same type (i.e. returned from the same component). The *BTL* API provides an abstraction over the underlying transport techniques, so where the component named sm (shared memory) is used to transfer data between processes on the same host using a FIFO shared memory construct, the tcp component would use socket file descriptors instead. This is completely transparent to both r2 and ob1.

A *BTL* module works with a simple tag and completion callback system. There

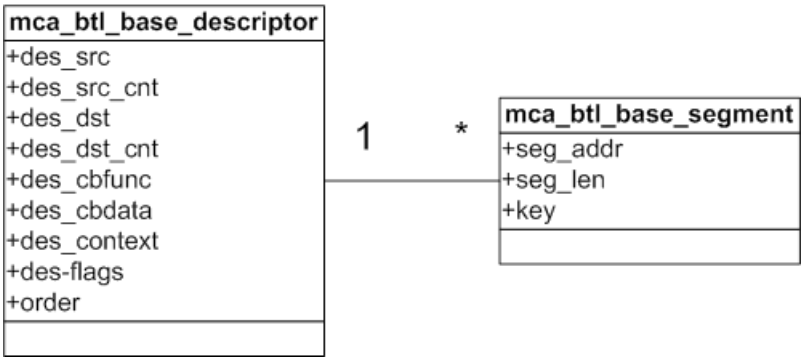

Figure 4.4: A *BTL* descriptor can have many memory segments

is an upper limit of 256 possible tags allowed (0 - 255) and currently very few of these are used. During the initialisation phase of a job, the *PML* through the *BML* will register various completion callback functions for specific tags. When a message is sent through a *BTL* module the higher layers will also pass a tag value that is separate from the message itself. It is then up to *BTL* module to transfer the data along with this tag and then module in the receiving process will match this tag to a completion function which it invokes to indicate the receipt of a message. This is a useful system as it provides a means for higher layers to separate and handle different messages in different ways while still ensuring the *BTL* module is oblivious to the nature of the messages themselves. As of Open MPI 1.3 `ob1` reserves 9 tags to use with the *BTL* modules.

A *BTL* module passes information around using descriptor and segment structures defined by the *BTL* framework. A descriptor structure `mca_btl_base_descriptor` contains an array of source or destination segments, message flags and a completion callback function with associated data. As of Open MPI 1.3 there is an extra field called 'order' which is optional at the moment, it will be assumed to have a default value and will not be used in the scope of this project. The segment structure `mca_btl_base_segment` is Open MPI's answer to an `iovec` structure; it contains a pointer to an address in memory, a length in bytes of the contiguous data at the memory location and a segment key which is only used in RDMA operations. Having an array of segment structures means we can work with segmented or non-contiguous data which will turn out to be very advantageous for this project. When data is to be the `des_src` field is used and when data is to be received the `des_dst` field is used, however they both simply point to an array of segment structures. See figure 4.4.

A *BTL* component/module has 4 stages in its life cycle. See figure 4.5.

1. The initialisation phase calls the component's open and initialize functions and allow modules to be created. This is called during `MPI_Init()`.

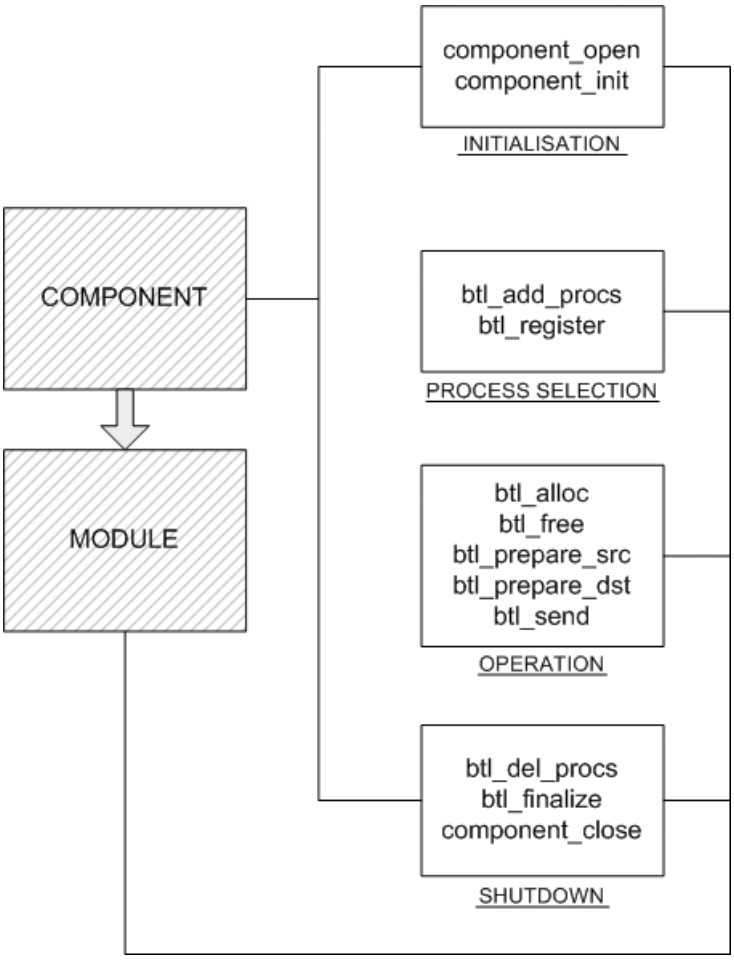

Figure 4.5: The four stages of a *BTL* component/module

2. The process selection phase determines which processes are reachable using the module. Endpoint structures are usually created in this phase too. This is also called during `MPI_Init()`.

3. The operation phase sends and receives raw data to and from other processes.

4. The shutdown phase releases any resources taken by the modules and component. This is called during `MPI_Finalize()`.

A standard *BTL* module has a public interface outlined in table 4.2. It may be noticeable from the API that a *BTL* module supports both a standard send/recv system and also a remote direct memory access (RDMA) system. A *BTL* module can choose either method or both. RDMA is out of the scope of this project and will not be discussed further; for our *BTL* module these function pointers will simply be set to `NULL`. The `btl_sendi()` function is new to Open MPI 1.3 and won't be used in the scope of this project and so it will also be set to `NULL`. This is perfectly safe as all these functions are not needed to create a functioning module.

It may also be noticeable that there is no function to progress the module. This is actually placed in the *BTL* component code instead of individual *BTL* modules, it is assumed the component has enough information about each module in order to progress them. For the purposes of this project other techniques are used to ensure messages are sent and received properly so the progress function pointer is set to `NULL` too.

Function	Description
btl_add_procs	Called to discover which processes are reachable by this module and create endpoint structures
btl_del_procs	Called to release resources held by the endpoint structures
btl_register	Called to register completion callback functions for a particular tag
btl_finalize	Called to release any resources held by the module
btl_alloc	Returns a *BTL* descriptor with free space but no actual data
btl_free	Releases a *BTL* descriptor and any memory used by it
btl_prepare_src	Returns a *BTL* descriptor that contains user data
btl_prepare_dst	Returns a *BTL* descriptor used for RDMA operations
btl_send	Initiates an asynchronous send of a particular *BTL* descriptor
btl_sendi	Initiates an immediate send of a particular *BTL* descriptor
btl_put	Initiates an asynchronous put (RDMA)
btl_get	Initiates an asynchronous get (RDMA)
btl_dump	Diagnostic information

Table 4.2: Functions available in a *BTL* module

One thing that may not be obvious from looking at the *BTL* module API is how receiving data should work. The only thing mentioned thus far is registering callback functions with an associated tag. The answer is that there is no standard technique and entirely it's up to the programmer to find the most appropriate way of achieving this based on the underlying technology. For example the `sm` (shared memory) module relies on its component's progress function being called periodically to check for changes in the shared FIFO structures. On the other hand, the `tcp` module relies on periodic polling of any open TCP/IP socket file descriptors.

The `btl_add_procs()` function is one of the more elaborate functions defined by the API. Its input parameters take in a list of all processes in the Open MPI job during the initialisation phase and its purpose is to identify which of these processes are reachable using this module. It is also used to create a *BTL* endpoint data structure for each reachable process. Both of these tasks can cause some confusion to programmers new to the *BTL* framework. With regards to the *BTL* endpoint data structure, the *BTL* framework does not actually define one, instead

it uses an incomplete structure declaration and allows the programmer to define it however they wish. This way the higher levels can cache pointers to the *BTL* endpoint data structures without ever utilising them. This proves to more efficient as higher layers can call the *BTL* functions and explicitly pass in *BTL* endpoint data structure as parameters instead of forcing the *BTL* module to match a process structure to a corresponding endpoint structure.

With regards to the other issue of identifying which processes are reachable via the module, we must introduce an Open MPI feature called the *OpenRTE Group Communications (grpcomm)* which is actually an MCA framework within the Open Run-Time Environment (ORTE) layer of Open MPI. Programmers who worked with Open MPI 1.2 and prior will be familiar with the *General Purpose Registry (GPR)* which had been used for a similar purpose. The *grpcomm* modules implement methods targeted to allow processes to communicate with large collections of other processes in specific communicator groups (e.g. `MPI_COMM_WORLD`). It operates out-of-band and how it establishes connections needn't concern the programmer. One must acknowledge these out-of-band channels will never be as efficient or as effective as the point-to-point channels established by a *BTL* module and should only be used for exchanging meta-data.

The *grpcomm* modules provide functions to help scatter metadata to communicator groups; however, Open MPI gives us some helper functions in the OMPI layer to make this slightly more transparent, these are known as the module exchange or modex functions. During the initialisation phase of an Open MPI job, when all components are opened and modules are returned `ompi_modex_send()` can be used to scatter metadata about the module. This would generally be addressing-type information, e.g. the `tcp` modules use this to exchange port numbers of their sockets. During the process selection phase there is a guarantee that the scatter will have happened by this point and consequently `ompi_modex_recv()` can be called on a particular process to receive the metadata. What's good about this is that somewhere between the initialisation phase and the process selection phase the ORTE layer will perform an out-of-band process information exchange anyway; if any additional metadata can be provided, it is sent essentially cost free.

In the case of trying to perform `ompi_modex_recv()` on a process that simply doesn't use the same kind of module (e.g. for a cluster of machines connected to each other over InfiniBand connections and also some additional machines connected using Ethernet, the machines lacking InfiniBand hardware would not use the `openib` component), it will return immediately with an error code and thus it is known the process can't be accessed the using this module. In the case of processes which use the same kind of module but are not suitable to communicate with each other (e.g. two machines connected using Ethernet each running two processes that communicate with each other using the `sm` shared memory component. Every process uses the `sm` component yet it isn't feasible to use the module between processes on different machines), the programmer is expected to provide enough

metadata in their `ompi_modex_send()` to determine the reachability of processes.

A *BTL* component can be fine tuned at runtime using MCA parameters. When an Open MPI job is launched, the user can optionally provide additional arguements as command line parameters. During the initialisation phase of a *BTL* component, there should be code to check to see if any of these parameters are set and to configure the component/module based on their values (otherwise defaulting to empirically optimal values). This can be useful for any number of situations, for example, offering the user the ability to define the maximum size of a message can potentially allow them to optimise the job for a particular application.

4.1.4 The Event System

Open MPI contains its own internal event system based on Niels Provos' libevent[18], it is located in the OPAL section of code. The event system is responsible for monitoring open file descriptors, scheduling timed callbacks and catching system signals. The programmer writing a component can utilise the event system to register callback functions that will be invoked when an event happens. The event system is compatible with file descriptors that implement the `select()`, `poll()` or `epoll()` functions. The event system can efficiently poll many file descriptors in the background for `POLLIN` or `POLLOUT` flags and will invoke the user's callback function on detection.

4.1.5 Object Orientation

An object oriented system can improve the ease of development and readability of code. Open MPI uses a single inheritance C-style object oriented model. Because different systems have different memory organisations for C++ objects, C was chosen as the implementation language instead and an object oriented utility system was implemented in the OPAL section of code.

Open MPI's single inheritance model can give the programmer a flexible, efficient and safe environment to work in. It is common practice to extend the structures defined in the *BTL* API using this model. For example, `mca_btl_base_descriptor` structures are created in the functions `btl_prepare_src()`, `btl_prepare_dst()` and `btl_alloc()`. There is a lot of extra metadata and context information that can be helpful to the programmer and yet doesn't fit in any of the given fields of the `mca_btl_base_descriptor` structure. The programmer can create a new structure that is inherited from `mca_btl_base_descriptor`. The new structure contains a `mca_btl_base_descriptor` variable as its first element and when returning from one these functions, the programmer would return a pointer to the first element of this new extended structure. When other *BTL* functions

get passed back this pointer, e.g. `btl_send()`, they can cast the pointer from the supertype to the subtype. This way higher layers see the exact same memory layout for the structure and are oblivious to the extra information, but lower layers can safely access the additional fields (provided they can guarantee the structure is subtyped).

The above doesn't have the full ease of object orientation; to remedy this Open MPI has its own system in which the programmer can utilise constructors, destructors and an object reference counting system. A class is made as described above using a structure with the parent type as its first element (the parent type must also be a class). In order to transform this structure into a class the programmer uses the macro `OBJ_CLASS_DECLARATION(NAME)` passing the structure type as an argument. The programmer can then associate a constructor and destructor for the class by using the macro `OBJ_CLASS_INSTANCE(NAME, PARENT, CONSTRUCTOR, DESTRUCTOR)` where `NAME` is the struct type, `PARENT` is the supertype's structure type and `CONSTRUCTOR` and `DESTRUCTOR` are names of the functions for the desired constructor and destructor (these must take a single parameter which is a pointer to a structure of the class's type). In order to use the class system, the programmer would use `OBJ_NEW(type)` which will create an object on the heap, initialise it using the constructor and return a pointer to it. If the object is to be used in several places around the system, the programmer can optionally use `OBJ_RETAIN(object)` which increments the object's reference counter. Similarly `OBJ_RELEASE(object)` decrements the reference counter and when it reaches zero, the destructor will be invoked and the memory freed automatically.

4.2 Introduction to the Xen architecture

Xen as a whole has a very complicated architecture that is best described by [8] and [1, p.27-46]. There are a subset of features that were specifically important to this project and these will be described in detail.

4.2.1 Hypercalls

Generally operating systems are designed specifically to run on processors with varied modes or privileges. This means that the operating system would run on the highest privilege mode of the processor and any processes it created would run in a less privileged mode. Consequently this means that a user process cannot execute operating system code directly as these routines require the processor to be running at a certain privilege or mode. Instead, the user process has to passively invoke these functions. This means lodging a request in a known destination of memory or a register and explicitly triggering an interrupt causing the operating

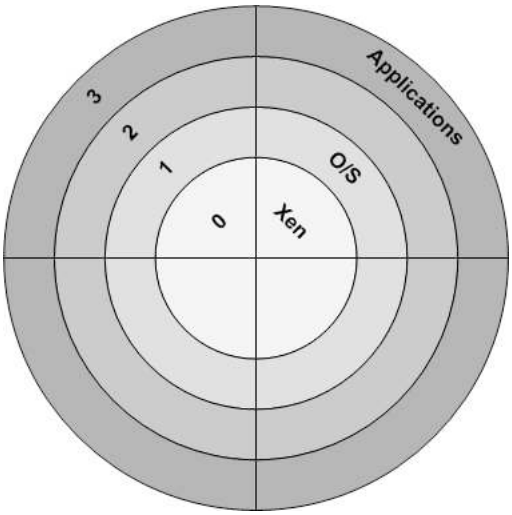

Figure 4.6: The operating system is moved to ring 1

system to schedule in place of the process. The operating system, running in privileged mode, then performs safety and security checks on the request and runs it on behalf of the user process before scheduling it back in. This is commonly known as a syscall and is a clever way to ensure processes remain isolated and don't execute code that could be detrimental to the rest of the system.

For Xen to operate correctly the scenario becomes more complicated. The Xen hypervisor has to provide the same isolation and security to all its virtual machine guests which in turn expect at least two privilege modes each (i.e. privileged for the operating system code and unprivileged for the user code). Therefore a processor with at least 3 unique privilege levels is required. Fortunately for the x86 architecture there are two solutions.

The x86 architecture is unusual in that it doesn't offer standard 2 level privileged/unprivileged modes, instead it offers 4 unique privilege levels known as rings. Traditionally the operating system has always occupied ring 0 and the user applications occupied ring 3 leaving rings 1 and 2 unused. Xen can take advantage of this and run in ring 0 allowing its guest operating systems to run in ring 1. This, however, requires modification of the operating system code to run in ring 1 instead of ring 0. For a Linux based operating system this isn't a problem as the kernel is open source. See figure 4.6.

The alternative comes in the form of hardware extensions offered by the two major x86 manufacturers Intel and AMD; the technology is known as Intel VTX and AMD-V respectively. Instead of requiring the operating system to be modified in order to run in a lower privileged ring, an extra ring called -1 is added to the processor which inherently has a higher privilege than rings 0,1,2,3. This way the Xen hypervisor can run in ring -1 and the guest operating systems can remain in

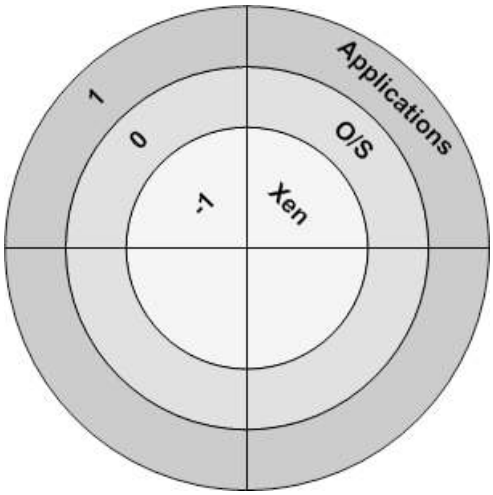

Figure 4.7: The operating system remains in ring 0

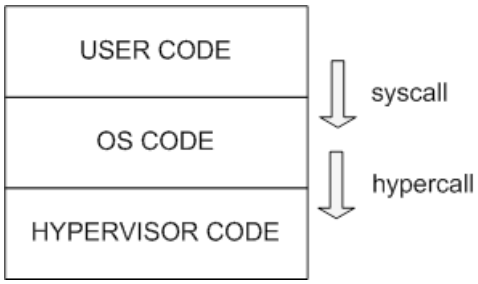

Figure 4.8: The guest operating system makes hypercalls to the hypervisor

ring 0. This is especially important for the x86-64 architecture which removed the 2 unused rings from its processors. See figure 4.7

This is important because if a user process needs to invoke a syscall in order for the operating system to perform sensitive work on its behalf, a guest operating system must have to do the same same thing with the Xen hypervisor. After all, for any device drivers in a guest operating system they can't directly communicate with the hardware so they must rely on another part of the Xen system to handle requests on their behalf. The solution is called the hypercall[2, p.47-74]. A hypercall is to the Xen hypervisor what a syscall is to an operating system. See figure 4.8. There exists a number of Xen hypercalls that can be used in the operating system's kernel space with a variety of uses, however the scope of the project only concerns itself with a few in particular.

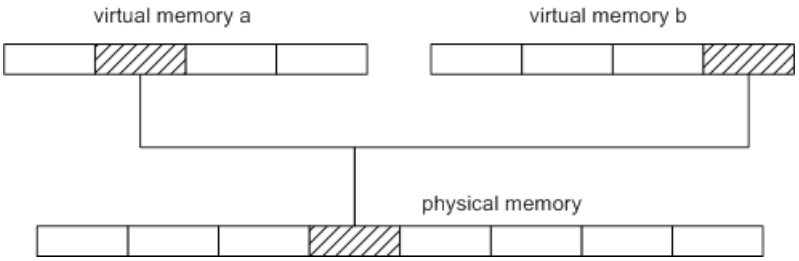

Figure 4.9: Virtual memory can be mapped to the same location

4.2.2 Grant Tables

Sharing memory between multiple processes on an operating system has generally been considered as a simple and efficient way to achieve interprocess communication. One process would allocate a pool of memory and then give access to it to another process on the system. It remains a secure model since it is only ever the process that owns the memory that can grant and revoke access to it to the other processes. Internally it is very efficient since every process has its own unique view of the system's memory known as virtual memory. The virtual memory space of a process maps onto arbitrary locations of the system's physical memory. This means that two processes can write to location 0x1000 of their own memory without interfering with each other as this location will be mapped to two unique locations in physical memory. When two processes wish to share the same memory, it remains as simple as mapping both of the process's virtual memory location to the same physical memory location. See figure 4.9.

Xen offers an analogous system to share memory between two isolated operating systems. Grant Tables[1, p.59-74] provide a mechanism for virtual machines to offer their memory pages to other virtual machines and then similarly for other virtual machines to map these pages into their own memory space. Mapping can only take place at a page granularity (usually 4Kb). One must do this in the virtual machine's kernel space and then use the virtual machine's operating system's constructs to bridge the user space to the kernel space.

In order to grant another virtual machine access to pages, the functions in listing 4.1 should be used.

```
int gnttab_grant_foreign_access(domid_t domid, unsigned
   long frame, int readonly)

int gnttab_end_foreign_access_ref(int ref, int readonly)
```

Listing 4.1: Code to grant memory to another virtual machine

Where `domid` is the unique identifier of the other virtual machine; `frame` is

the machine frame number of page(s); `readonly` is a flag to indicate if the other virtual machine is allowed to write to the pages or not. `gnttab_grant_foreign_access` returns an integer known as a grant reference and `gnttab_grant_foreign_access` returns 1 if the pages were granted successfully or 0 otherwise. One can use `int virt_to_mfn(int va)` to convert a virtual machine's heap address into the machine frame number.

In order for the other guest to map and unmap pages, functions in listing 4.2 should be used.

```
int HYPERVISOR_grant_table_op(unsigned int cmd, void *
   uop, unsigned int count)

struct gnttab_map_grant_ref
{
        /* IN parameters. */
        uint64_t host_addr;
        uint32_t flags;
        grant_ref_t ref;
        domid_t  dom;
        /* OUT parameters. */
        int16_t  status;
        grant_handle_t handle;
        uint64_t dev_bus_addr;
};
```

Listing 4.2: Code to map memory from another virtual machine

For the `HYPERVISOR_grant_table_op` function, `cmd` should be the constant `GNTTABOP_map_grant_ref`; `uop` should point to an array of `gnttab_map_grant_ref` structures and `count` should be be the amount of elements in the array.

For the `gnttab_map_grant_ref` structure, `host_addr` should point to somewhere in the virtual machine's heap allocated by `alloc_vm_area`; `flags` should be the constant `GNTMAP_host_map`; `ref` should be the grant reference returned from the granting function and `dom` should be the unique id of the virtual machine that is granting us access to these pages. `status` will be non-zero if an error occurred and `handle` is a structure used to help unmap these pages at a later stage.

To unmap these pages we call `HYPERVISOR_grant_table_op` again changing the parameters slightly. `cmd` should be the constant `GNTTABOP_unmap_grant_ref` and `uop` points to an array of `gnttab_map_grant_ref` structure taking the following form: `host_addr` should point to the location in the heap where the pages are mapped to and `handle` should be the structure generated when the pages were initially mapped. The remaining values can be zeroed.

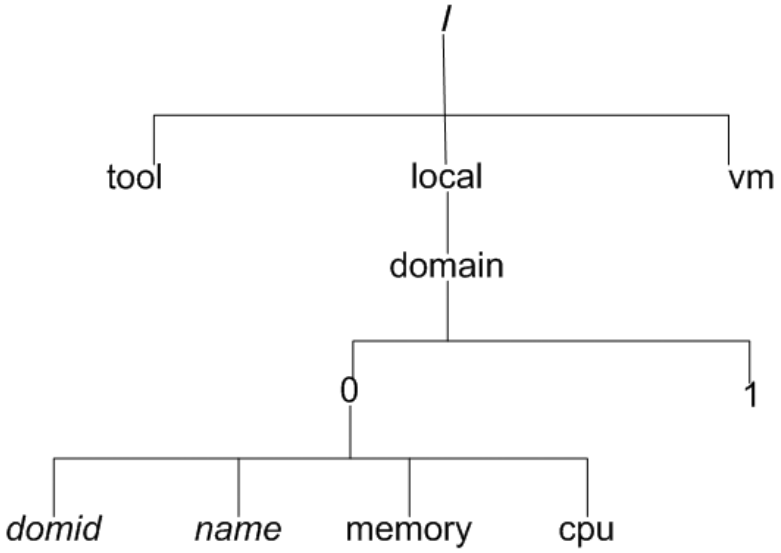

Figure 4.10: An example layout of XenStore

4.2.3 XenStore

Sharing memory between virtual machines is a very useful system to have, however it requires a certain amount of knowledge about the virtual machines ahead of time. For example, granting another virtual machine access to pages of memory requires that the virtual machine's unique identifier (domid) is known. Each virtual machine is therefore expected to be able to find out their own domid somehow. Fortunately this value, along with a wealth of other information, is stored in a database known as XenStore[1, p.141-160].

XenStore is a tree-like database with key-value pairs. There is a root with a mixture of branches and keys, within these branches there are more branches and keys. The kind of information stored in XenStore would be metadata about the virtual machines themselves. For example: their names, unique identifiers (domid & uuid), memory allocated or the number of virtual CPUs allocated. To navigate through XenStore one would list the branches and keys located at a particular location (e.g. root is /) and then in order to read a particular key, one would format their location to something like */local/domain/0/domid*. See figure 4.10.

One issue that may seem to exist in the database shown in figure 4.10 is that one needs to know the domid of the virtual machine in order to navigate to its branch in the tree. Much like a UNIX filesystem, when interacting with XenStore and the root argument / is missing from a location, it will assume the current directory instead. By default, the current directory is actually */local/domain/your_domid/* so one would merely need to query *name* in order to find the name of their virtual machine.

XenStore is actually an application running in dom0 and has very little to do with the underlying hypervisor. Each virtual machine can access XenStore through a virtual device driver, however there is a well defined interface in the file (`xs.h`) which can be included in user applications in order to interact with it.

By default there is a strict security model for XenStore. For example, dom0 can read and write to all the branches of the database and grant and revoke access for other virtual machines to read or write to particular branches of the database too. By default a virtual machine can read and write to the to its home directory and its sub-branches. Care must be taken however, as in the default security model for guest virtual machines, only root users can interact with XenStore. This can be changed by modifying the access privileges of the virtual device, in Linux the device is */proc/xen/xenbus*.

5 Design and Implementation

Section 4 examined some of the Open MPI architecture and how the *BTL* framework is used to make point-to-point connections between processes. It also described some of the constructs offered by the Xen API that facilitate creation of alternative communication channels between virtual machines. This section describes how an Open MPI *BTL* component that utilises these Xen constructs was designed and implemented. This component will be referred to as `xen`.

One of the critical design decisions lay in the fact that Open MPI is a user space library and the Xen API works in kernel space only. This meant that the `xen` component could not be entirely cohesive; there would have to be code written in kernel-space to execute memory mapping operations and set up any inter-VM event channels, there would also need to be code in user-space to co-ordinate and interact with this kernel code.

Section 3 described some technologies that attempted to remedy the problems of Xen's inter-VM communication system. One of these was XenSocket, a light weight loadable kernel module that offers a unidirectional direct path between two virtual machines by granting/mapping shared memory. XenSocket bridges the kernel-space/user-space division by using a standard Berkeley socket interface. The disadvantage of XenSocket is that it inhibits any form of non-blocking interaction and has no way to notify user-space applications when data is available. Another problem of XenSocket is that it had been written and released in 2007 and had not been updated since then. When attempting to compile it on a more recent Xen hypervisor and Linux kernel, it was found that many of the constructs and functions were deprecated or changed.

A decision was made to build a *BTL* component that would use a pair of XenSocket file descriptors, one for sending and another receiving. The only way XenSocket in its default state could work with Open MPI would be by introducing threads into the Open MPI component. For every receiving XenSocket file descriptor created a thread would also need to be created to continually perform a blocking receive operation on the socket. [17] discusses why this is not a optimal solution, especially if the goal is efficiency. Additionally, this method doesn't fit in with Open MPI's best practice model. In order to use the good features of XenSocket with Open MPI, some additions and changes would need to be made to it.

5.1 A Custom XenSocket

Before customising XenSocket, it first had to be updated to work with Xen 3.2 and the Linux kernel 2.6.25. This was achieved by investigating how socket features

are implemented in [3] and how grant tables and event channels work in [1]. The rest was achieved by looking at Linux kernel code and Xen header files and also examining the Xen changelog. The update was successful; the new source code looked semantically the same as the original and the module itself performed as expected. It has also been demonstrated to work correctly on Xen 3.3 and the Linux kernel 2.6.27.

The next step was implementing functionality to notify the user-space code when there is data in the XenSocket buffer. Two methods were tried. The first was a real-time signal system. Linux real-time signals allow a running process to be interrupted by invoking a system signal; the running process catches this signal by using a signal handler. The benefit of real-time signals is that they allow an additional piece of information to be sent to the user's signal handler. The goal was as follows: whenever a XenSocket was sent data, it would notify the receiving process with a signal and additionally send the unique id of the socket file descriptor to the process's signal handler. This way the signal handler could work with many sockets without performing any additional checks to determine which file descriptor invoked the signal. For reasons outlined in section 7 this design did not function correctly and so another method was tried.

Section 4 specifies the Open MPI event system which can be used to monitor file descriptors by using the `poll()` and `select()` functions. The event system can watch for the `POLLIN` and `POLLOUT` flags in large number of open file descriptors concurrently. [18] shows that the time taken to poll multiple file descriptors rises linearly; so for this reason and also for the fact that there will be a rational upper limit on the amount of virtual machines hosted on one physical machine, a polling system was implemented. The standard Berkeley socket interface has a place holder for a select/poll function. Some of the code for the function was taken from other socket implementations found in the Linux source code. XenSocket keeps a shared structure that contains values such as the size of the buffer and the amount of data currently in the buffer. The polling logic simply looks as these values; if the amount of data in the buffer is greater than 0 it will set the `POLLIN` flag; if the amount of data in the buffer is less than the size of the buffer it will set the `POLLOUT` flag. This requires minimal code and can inform the user application when it is suitable to read and write data to and from the socket.

Two separate designs of the *BTL* component were developed. One design reads a known amount of bytes from the socket in order to retrieve the header of a message. The header contains a field with the size of the rest of the message so the socket could then be read again for this amount of bytes. Because this system used two system calls to retrieve a single message, an additional design was developed to remedy the issue. Instead of trying to retrieve one message in two steps, the socket is queried to see how much data is pending in the buffer and consequently this amount of data is retrieved into user-space in one go. It is then up to an algorithm in users-space to filter out complete messages. Doing

this requires some way to find out how much data is in the XenSocket buffer. To achieve this, the socket's IOCTL functionality was extended to accommodate the request. XenSocket was subsequently given one command to detect the amount of free space in the XenSocket buffer and another to detect the amount data pending in the XenSocket buffer.

5.2 The xen *BTL*

The xen *BTL* was designed to work with a pair of XenSocket file descriptors, however the actual design had three different incarnations. Two of these worked correctly and the other was faulty and so abandoned early in the project life cycle. Of the two which worked, it is interesting to note that the design unlikely to yield the best performance actually outperformed the other for larger messages. The two designs that operated correctly share a common groundwork which will be described in detail; their unique features will be also be described and then analysed in section 6.

5.2.1 The Initialisation Phase

The first phase of the *BTL* life cycle focuses primarily on searching for an appropriate environment to run in and allocating and initialising resources appropriately.

The xen component structure was the first item implemented; its task is to register parameters and detect if the environment is suitable to run in and returning a module instance conditionally. As discussed in section 4, there are four main functions in a *BTL* component: open, close, init and progress. The progress function is called periodically in order to help move data in intervals. It was decided from the beginning that because the component/module would work with sockets, other techniques would be used to move data to and from endpoints. Therefore the xen module does not have a progress function.

The component is also the structure that houses customisable fields that fine tune the module's run-time behaviour. The fields in table 5.1 were used to customise the component. The `btl_xen_free_list_*` fields are best described in context and are explained later this section. `btl_xen_buffer_order` is used to adjust the size of the underlying buffer in the XenSockets; if buffer order is x then the buffer size will be $2^x * PAGE\_SIZE$. Care must be taken here as the Linux kernel can only guarantee (if the memory is available) an order of 10 or 11 (corresponding to 1024 or 2048 pages). `btl_xen_exclusivity` refers to the priority of using the xen component over another component, it is defaulted to the constant `MCA_BTL_EXCLUSIVITY_DEFAULT` which means it will be prioritised over the tcp module which is set to `MCA_BTL_EXCLUSIVITY_LOW`. `btl_xen_eager_limit`

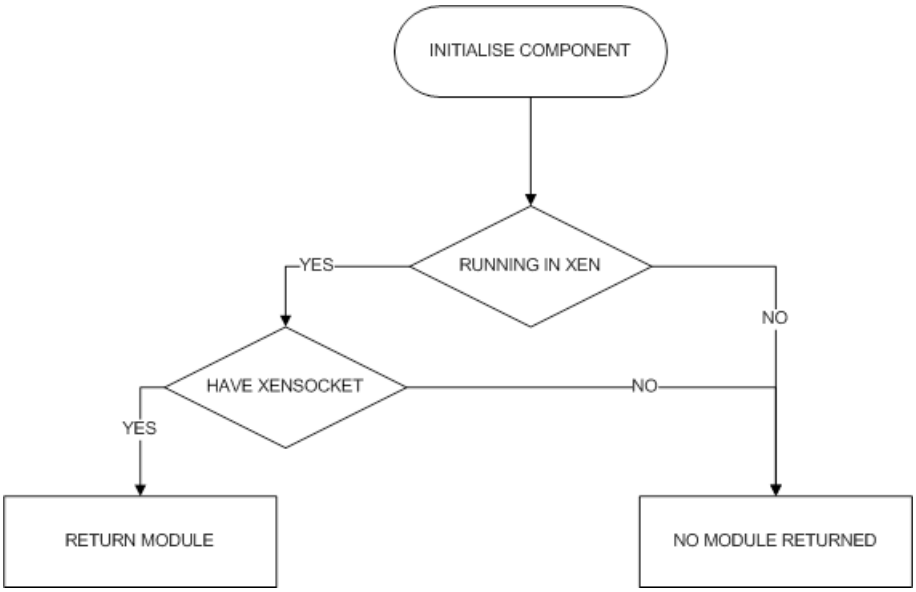

Figure 5.1: The xen component checks for a suitable environment to run in

refers to a maximum size of a short message, we can set this to improve internal memory management. `btl_xen_max_send_size` is the maximum possibly size of a message that can be sent before it needs to be fragmented.

During the xen component's initialisation routine it is required to detect if it is running in a Xen paravirtualized environment and subsequently check for the presence of the XenSocket loadable kernel module. This was accomplished with some example code found in the Xen package, it contains a `cpuid` assembly instruction to detect a paravirtualized environment. Checking for the presence

Field	Description
btl_xen_free_list_num	The number of elements initially in a list
btl_xen_free_list_max	The number of elements a list can grow to
btl_xen_free_list_inc	The number of elements to add when growing a list
btl_xen_buffer_order	A parameter that decides on the size of the XenSocket internal buffer
btl_xen_exclusivity	Can change the priority of the component during a selection phase
btl_xen_eager_limit	A probable size of a message going through a XenSocket
btl_xen_max_send_size	The maximum size of a message going through a XenSocket

Table 5.1: Customisable parameters of the xen *BTL* component

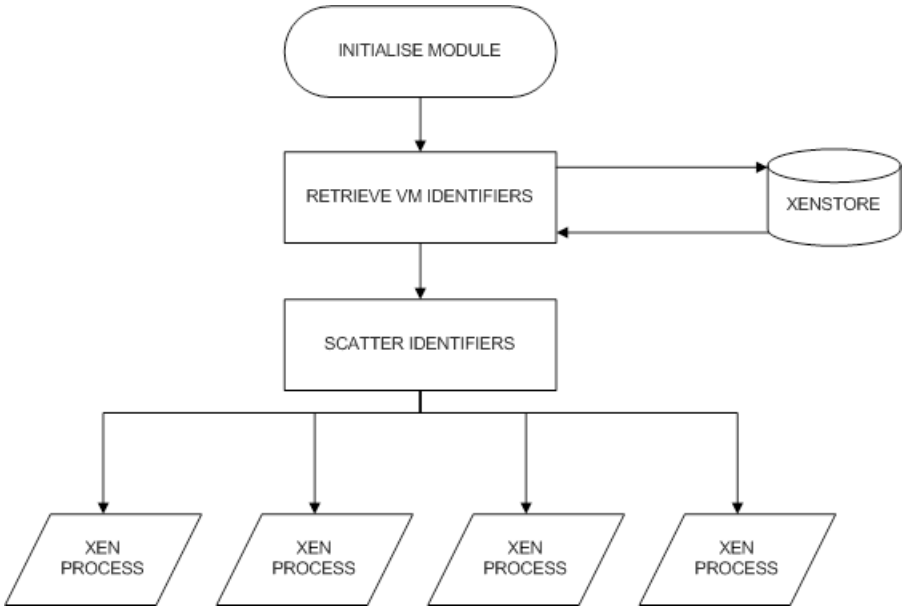

Figure 5.2: A `xen` module scatters its metadata to other `xen` modules

of XenSocket was achieved by scanning the contents of */proc/modules* (this is user readable by default). Provided these checks pass a single `xen` module can be allocated and returned, see figure 5.1. When initialising the module, it will access XenStore to find the domid and uuid of the virtual machine it is running on. After retrieving these values it will perform a module-exchange using `ompi_modex_send()` to scatter these values to every process in the job. See figure 5.2.

5.2.2 The Process Selection Phase

The second phase of the *BTL* life cycle requires finding processes to interact with using the `xen` module and establishing communication endpoints with them. In the initialisation phase, the process performed a `ompi_modex_send()` in order to scatter its virtual machine metadata to every process using a `xen` *BTL* module. This is the key to determining which processes are reachable in our job. When `btl_add_procs()` is called, the higher layers pass in an array of `ompi_proc_t` structures as a parameter. These structures don't contain any useful data for the `xen` module directly but they do provide a field to allow calling the corresponding `ompi_modex_recv()` function.

Inspired by other *BTL* implementations, the process selection phase is split into two stages and part of the process detection has been placed in a separate area of code. In essence, there is a structure called `mca_btl_xen_proc_t` which contains a pointer to the original `ompi_proc_t` structure along with the additional virtual machine metdata. A helper function called `mca_btl_xen_proc_create()` is

defined which that takes the original ompi_proc_t structure as an argument and attempts to create and return a matching mca_btl_xen_proc_t instance. Internally this works by calling the ompi_modex_recv() on the process structure to retrieve this virtual machine metadata. As described in the initialisation phase, if any process in the job has opened a xen *BTL* component and successfully returned a xen *BTL* module it will call ompi_modex_send() in order to send its domid and uuid. This is exactly the information we get when ompi_modex_recv() is called. For a process that did not successfully obtain a xen *BTL* module, calling ompi_modex_recv() with it as a parameter will fail gracefully and no corresponding mca_btl_xen_proc_t instance will be created. To ensure this process doesn't happen multiple times, a hash table is used to store previously created mca_btl_xen_proc_t instances using the original ompi_proc_t structure's unique name attribute as the hash key. The function quickly checks the hash table before attempting to call the ompi_modex_recv() function. If a mca_btl_xen_proc_t instance cannot be made or found, NULL is returned.

Within btl_add_procs(), the function iterates over an array of ompi_proc_t structures and calls mca_btl_xen_proc_create() on each one. If NULL is returned, it is known that the process isn't utilising a xen *BTL* module. If a mca_btl_xen_proc_t instance is returned, some additional checks must be performed. In section 4 it was mentioned that there could be circumstances where a process is definitely utilising a particular kind of module, but it still considered unreachable. Consider the scenario where there are two machines connected over an Ethernet connection; each physical machine is running a Xen hypervisor and houses 2 virtual machines each: A1 & B1 on machine 1 and A2 & B2 on machine 2. They all successfully get a xen *BTL* module and within btl_add_procs() they will all successfully return mca_btl_xen_proc_t instances for all four processes. The problem here is that although each process is using the a xen module, only processes on co-resident virtual machines can establish endpoints with each other. It makes no sense to attempt to establish endpoints between A1 and B2.

This is where the metadata becomes useful. The process should ask Xen if the virtual machine identified by the other process's domid and uuid fields exists on the same hypervisor. To do this requires accessing XenStore and requesting a listing of the entries at */vm*; this will give a list of all the uuids of co-resident virtual machines. The reason to use uuid instead of domid is because uuid is a GUID to identify a virtual machine universally whereas the domid is simply a integer starting from 0 that identifies virtual machines locally. To achieve this, xs_directory() is called to get an array of strings at the directory mentioned above of XenStore. All that is required then is a simple string comparison against each entry in the array. One final precaution is checking to see if the process exists in the same virtual machine to us (as opposed to another virtual machine on the same hypervisor). Because Open MPI won't discriminate this by itself, we are forced to perform this check. While it would be harmless to add these processes to the list of processes reachable through the xen module, it is best practise to

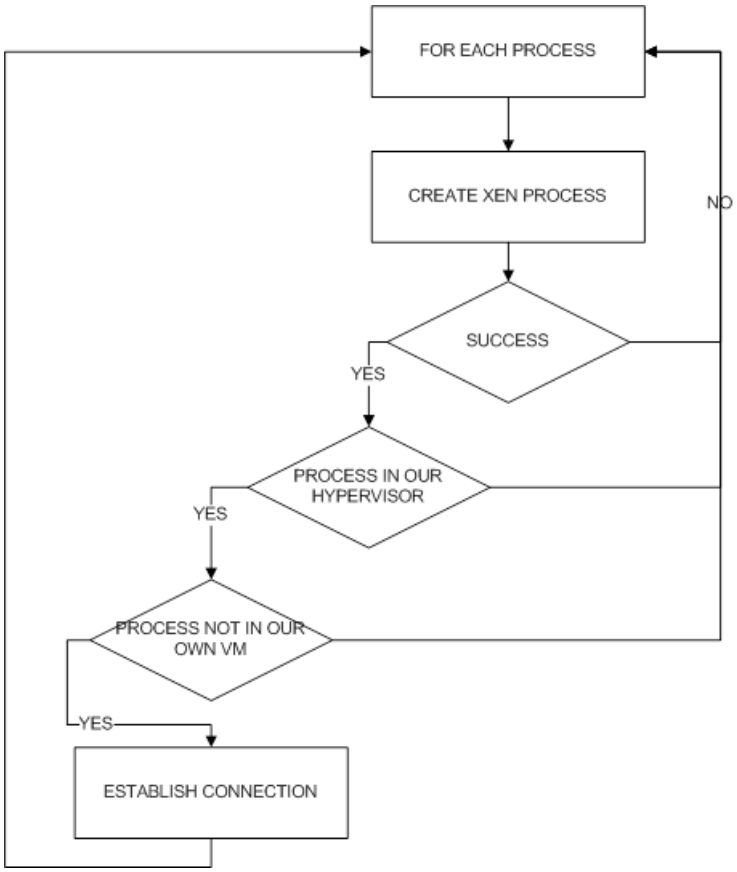

Figure 5.3: A xen module searches for processes to communicate with

avoid doing so. Open MPI has a *BTL* component specifically designed for inter-process communications and there is little point in wasting resources establishing extra unused endpoints. See figure 5.3.

For each process found that uses a xen *BTL* module and also detected to exists on a co-resident virtual machine, the function can proceed to establishing an endpoint. It was stated in the section 4 that an `mca_btl_base_endpoint_t` structure is undefined by Open MPI and is left to the *BTL* programmer to define it for their own custom needs. At the very minimum, a xen endpoint needs to contain two XenSocket file descriptors in order to send and receive data.

A XenSocket must first be created by the receiving side where a unique identifier known as a grant reference (gref) is returned. To create the receiving XenSocket, the domid of the virtual machine that is allowed to send the data must be known. This domid value can now be found in the `mca_btl_xen_proc_t` instance. The problem still remains of how to get the grant reference to the other virtual machine which is required to open the sending socket. To achieve this, an out of band channel established by Open MPI is used.

The function used is `orte_rml.send_nb()` where *RML* stands for *Remote Messaging Layer*. It is worth mentioning that this is actually a type of proxy to to another Open MPI utility called *OOB* that creates and utilises out of band channels. The *RML* is actually in charge of establishing, maintaining and routing messages through the *OOB* utility. Using this instead of an out of band channel directly is better practice and can prevent extra out of band connections being established unnecessarily. The `nb` suffix of the function indicates the send is non-blocking. The reason a non-blocking send is used in lieu of a blocking send is simply because there is no way to ensure the matching process can call a blocking receive. For example, if the matching process was already performing a blocking send to another process, the whole job would result in deadlock.

There is now a file descriptor for receiving data, however not for sending data. In order to do this there must by a corresponding *RML* call to receive the gref value sent via the out of band channel. Instead of explicitly doing this within the `btl_add_procs()` function, a decision was made to postpone this happening until later. This way the *BTL* can establish the other duplex of its connection on-demand when there is actually a necessity to send data. The first time data is attempted to be sent, a check is made to detect if the endppoint is fully established. If it's not, the corresponding `orte_rml.recv()` function is called and the grant reference is returned allowing the creation of the sending XenSocket. This technique allows some extra time to let the out of band send to propagate.

5.2.3 The Operation Phase

The operation phase of the *BTL* life cycle deals with the moving of data to and from our XenSocket pair. This is probably the most significant phase of the `xen` *BTL* life cycle as it is the area that affects throughput, latency and processor usage the most. The main design proved to outperform the `tcp` module which is shown in section 6, however there was also a variation of the design which outperformed the original. Because the differences are minor in contrast to the overall design, both are described.

When designing the method to send data, it had to be taken into account that a XenSocket uses a finite buffer which can become full and block the sending process. There is a symbiotic relationship between the two communicating processes as the only way a blocked send can proceed and return is if the matching process performs a receive and clears the buffer. A problem arises when both processes wish to send more data than there is available space in the buffers. They will both block, and because they are blocking there is no way for the processes to return and empty out the buffers so this results in deadlock. Naturally this problem extends to sending messages which are bigger than the buffer itself.

The way around this came in two forms. First, there had to be an upper limit

on the amount of data that can be sent through a XenSocket in a single send operation. This limit was naturally to be the size of the buffer or less and it was empirically found that an upper limit of $\frac{1}{8}$ the size of the buffer worked best. Secondly, there had to be a means to find the available space in the XenSocket buffer in order to decide if a message can be sent immediately or if should be scheduled for later a time.

In order to limit the size of single messages there had to be a system to allow the fragmentation of raw data on the sender's side and the reassembly of the data on the receiver's side. Fortunately there are higher levels in the Open MPI framework that can help achieve this very effectively. `btl_prepare_src()` is called by higher layers in order to construct a `mca_btl_base_descriptor_t` that contains some user data. The function has a variety of parameters including a reserve size, a `ompi_convertor_t` structure and and in/out integer called `size`. The reserve size is some space we must leave at the beginning of the message for the *PML* to place its header. The `ompi_convertor_t` is a helper structure used to point to the actual user data; the total size of the user data is then indicated by the in/out parameter `size`. There is a helper function called `ompi_convertor_pack()` that will copy x amount of bytes of the user's data into a specific location of memory. The in/out parameter `size` is then used to indicate to the *PML* exactly how much data was taken and placed into the allocated descriptor. If it's less than the size of the whole data the *PML* will indicate this in its header. When all of the user data is transferred using several descriptors, the entire message can be reconstructed in higher layers at the other end.

One of the benefits of the `ompi_convertor_t` argument is that there exists a helper function called `ompi_convertor_need_buffers()` which indicates whether or not the user data is contiguous or not. If it is contiguous, an extra copy can be avoided and by simply pointing to the data, otherwise another helper function is used to pack x amount of bytes into the descriptor's buffer.

Although there is reserve space for the *PML* header, it is vital that there is a custom header specific to the `xen` *BTL* too. This header contains 3 fields: the size of the message as a whole, the size of the *PML* header (i.e. whatever the reserve size was) and the tag of the message itself. The tag is not included in the *PML* header and is required by the *BTL* module to find the correct callback function during the receive stage. A structure called `mca_btl_xen_fraghdr_t` is defined in order to house these 3 fields.

`btl_prepare_src()` returns a descriptor to the higher layers of the framework which subsequently adds header information in the reserved space before calling the `btl_send()` function with the modified descriptor as a parameter. In Open MPI 1.2 and prior a *BTL* module's send function was for submitting a descriptor to a send queue and returning a success or error code. It was then up to the *BTL* module to dequeue the descriptor, send it, and invoke its callback function to tell

higher layers that the message was successfully sent. As of Open MPI 1.3 there is an alternative approach whereby if the *BTL* module finds it suitable to send the data immediately within the `btl_send()` function it can avoid invoking the callback function and instead return a code that indicates the data was sent via the shortest path. This is useful for the `xen` *BTL* module as a lot of the time there will be enough space in a XenSocket buffer to accommodate an immediate send.

The crucial decision in `btl_send()` is deciding whether or not to send the data immediately or scheduling it for a later time. It was mentioned previously that in order to avoid deadlock it must be certain there is enough space in the XenSocket buffer before sending a message. To find out the amount of free space in the buffer, IOCTL functionality that was added to XenSocket can be used. If it is found that there is enough space to pass in the whole message, a send can be performed immediately, otherwise the descriptor is added to an ordered list and scheduled to send at a later time. Adding the descriptor works by first checking the flags of the descriptor for the presence of `MCA_BTL_DES_FLAGS_PRIORITY`, if found the descriptor is prepended to the list, otherwise it is appended. In order to reschedule the send, Open MPI's event system is used. A request is made for a callback to be performed when the sending XenSocket is writable. In our custom XenSocket, there is a simple polling mechanism that indicates when there is data in the buffer and when the buffer has some space available. The XenSocket will be polled with all the other open sockets in the Open MPI job periodically so it can be guaranteed that the callback function will be invoked at some point. Within the callback function, the space available in the XenSocket buffer is first checked and then the function iterates over the list of descriptors in order to see if any can be sent without blocking.

All this relies heavily on the IOCTL functionality of the XenSocket which means invoking a system call every so often. Unfortunately system calls are very expensive due to their mechanism of going through libc and being context switched into the kernel, therefore it is desirable to keep them to an absolute minimum. IOCTL calls can be avoided by keeping a local record of free bytes available in the XenSocket buffer. For example, if the size of the buffer is found once and recorded, one can continually subtract the size of all successfully sent messages from this record. There is then a guarantee that *at least* that amount of space in the XenSocket buffer is available since the only other process effecting the free bytes in the buffer is removing data rather than inserting it. A record is kept in each endpoint structure of the guaranteed space available in the sending XenSocket buffer. If the descriptor can fit in this space there is no necessity to perform an additional IOCTL. An IOCTL call is only ever made whenever is is found that there is not enough guaranteed space to send the entire descriptor. Each time an IOCTL call is made, the record of the size of the XenSocket buffer is updated too. See figure 5.4.

Because messages can be in three separate memory locations (the `mca_btl_`

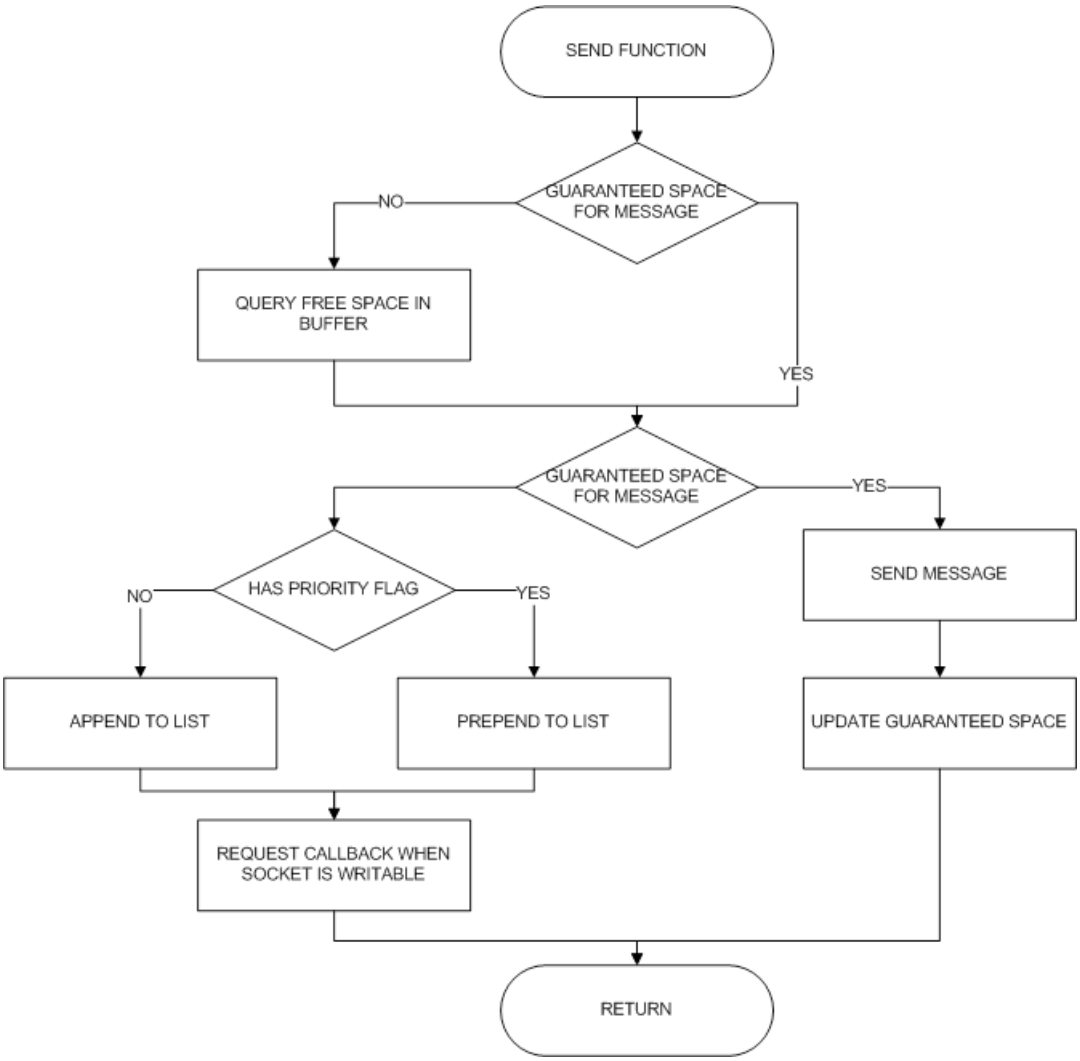

Figure 5.4: A xen module attempts to send a message

xen_fraghdr_t structure, the *PML* header data and the user data are in no way guaranteed to be in contiguous memory) a decision was made to send data using an array of iovec structures with the writev() function. This means all three memory locations could be handled in one single system call instead of three separate calls to the standard socket send() function.

Because there is no reliance on the *BTL* component's progress function, receiving data had to work another way. Open MPI's event system can poll file descriptors for pending data. This feature is used to invoke a callback function whenever the event system finds there is data in the XenSocket buffer. One of designs of the callback function used a circular buffer at each endpoint that was the same size as the XenSocket buffer. When the event library invokes this callback function it would first perform an IOCTL operation to find how much data is pending in the XenSocket buffer and then copy it all into the circular buffer. This was potentially a good idea as it allowed a maximum amount of information to be copied in a single step avoiding excessive system calls. After this receive is made, the raw data in the local buffer is sifted through to find complete messages.

Since the first thing transmitted in a message from the xen *BTL* is a mca_btl_xen_fraghdr_t structure, a check is made to see if the circular buffer contains at least sizeof(mca_btl_xen_fraghdr_t) bytes. If it does, a mca_btl_xen_fraghdr_t variable can be made from the raw data. Because this is a circular buffer, care must be taken to identify if data has been fragmented at the end and continued at the beginning. This is done by having variables for the offset of the buffer and the used space of the buffer, then finding it the data can be obtained in a single copy or if it is fragmented at the end in which case two copies are required. If an entire sizeof(mca_btl_xen_fraghdr_t) variable is obtained, a further check is performed to compare the size of the entire message (contained in one of the structure's fields) with the size of the used space of the buffer. If the used space of the buffer is greater or equal to the size of the message, the whole message can be retrieved, otherwise the callback function can finish gracefully knowing that there is still more data to come.

One of the quirks of the receiving callback function is that the *PML* header can't be fragmented. This is where knowing the *PML* reserve size in the mca_btl_xen_fraghdr_t structure can be very useful. The same technique as before is used to see if the header wraps around the circular buffer. If it does, the data must be copied into contiguous memory (this is fine since the data is relatively small) otherwise pointing to the circular buffer will suffice. The rest of the data is allowed to be fragmented so one or two segment structures can be used to point to it in the buffer rather than performing an additional copy.

A mca_btl_base_descriptor_t structure is made from the PML header and raw data. The tag field from the mca_btl_xen_fraghdr_t structure is used to find the corresponding callback function. It is invoked passing in the descriptor

which may have several segments pointing to different places in memory. This is perfectly legal and it is the *PML* module's responsibility to copy all this data into its own contiguous buffer. After the callback function returns, the position and size of the user buffer is updated to remove the data just processed. The process is repeated until there isn't enough data in the local buffer to process a complete message. At this point the callback function returns not to be called again until the event library finds more data in the XenSocket buffer and then the entire process is attempted again. See figure 5.5.

This design seemed like it would be the optimal way of retrieving and processing data as it could potentially pull many messages at once from the XenSocket buffer and thus eliminate repeated system calls to receive data. Surprisingly, a variant of this design performed better. Instead of the XenSocket polling feature being implemented by checking for data present in the buffer, an atomic counter was used instead. Each time a message was sent to XenSocket the counter was incremented and each time a message was pulled from XenSocket the counter was decremented. The fundamental difference in this design is that within the receiving callback function. Instead of querying how much data is in the XenSocket buffer and retrieving that amount, a single `mca_btl_xen_fraghdr_t` structure is first retrieved, from this it can be determined how long the subsequent data is and then remainder of the message is received. For the second receive a flag is used to indicate to XenSocket to decrement the atomic counter. This means two system calls are performed for each message and the callback function is limited to processing a single message only. The benchmarks in 6 show that the alternative design performs better for larger messages.

In order for the operation phase to be efficient, special care was taken in the memory management of the `xen` component. Since there is a high demand for the `xen` module to create descriptors for messages that may contain sparse amounts of data (e.g. the descriptor holds a synchronisation message with no user data) or bulks of data (e.g. if the user's data was non-contiguous it must be copied to a contiguous location of memory) there is a significant issue of allocating and deallocating memory effectively. The following techniques were based on other *BTL* component implementations and are described here for clarity.

Calling `malloc()` and `free()` are expensive operations and because of the frequency of messages created and returned to and from the *BTL* module it is best to avoid calling them as much as possible. Open MPI offers its own dynamic list construct called `ompi_free_list_t` as an alternative. Instead of allocating memory for individual messages using `malloc()`, a free item from a list is requested instead. The list can grow and shrink by a user defined amount so memory allocation and deallocation happens less frequently. When an item is returned back to the list it can be retained for the next request thus avoiding more memory allocation. The list construct fits in with Open MPI's object oriented system; each time an item is requested from the list it can be automatically passed to a user defined constructor

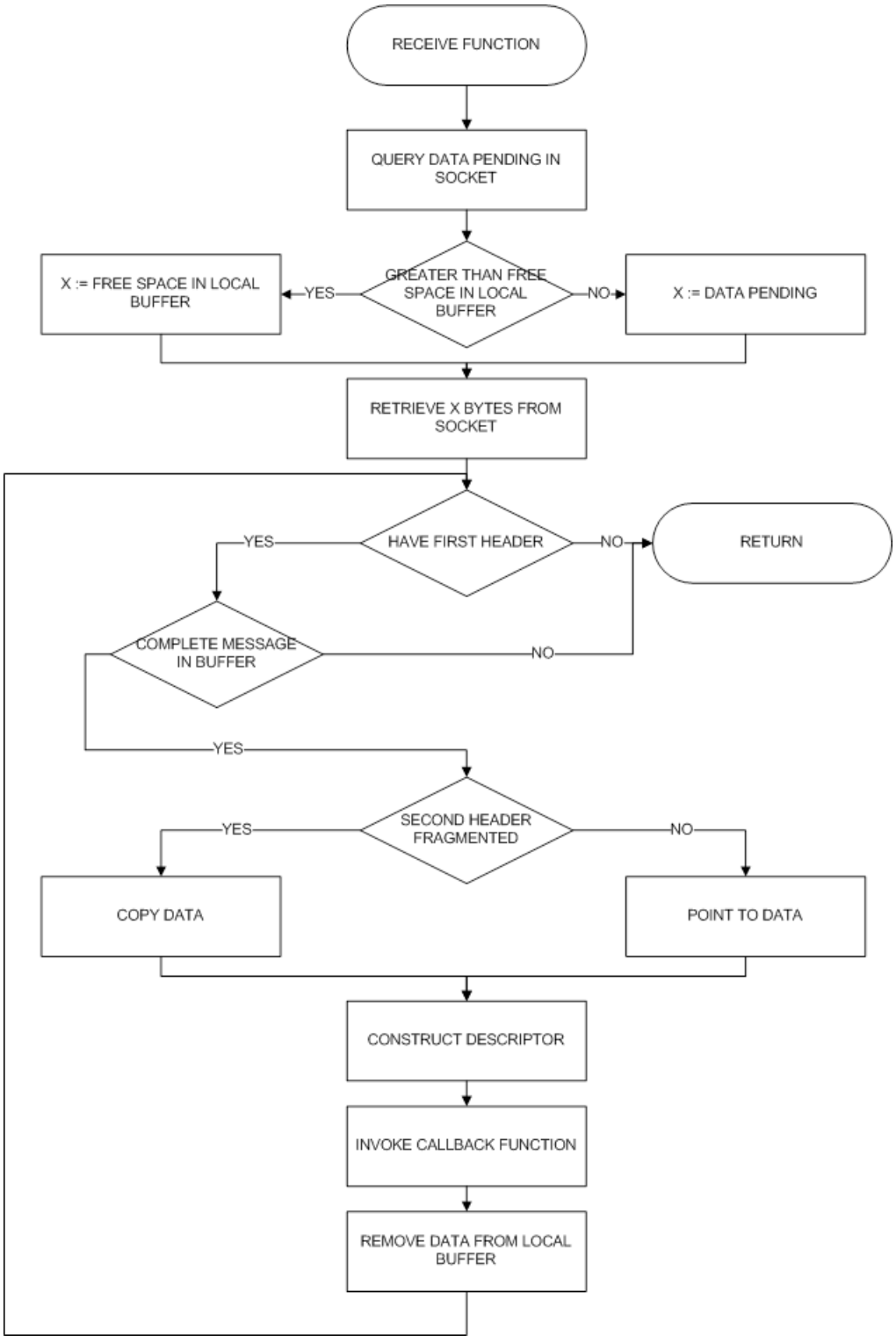

Figure 5.5: A xen module attempts to receive messages

where the fields are reset or defaulted to the programmer's preference. As of Open MPI 1.3 the `ompi_free_list_t` initialisation function has extra parameters to align the items in memory which can help improve caching.

The `xen` component has 3 such lists. One list is for returning a descriptor and memory for messages that only need space for the *PML* header (e.g. barrier messages or messages where there is no requirement to copy the user's data as it is contiguous); another is for returning a descriptor and memory for messages that need to be explicitly copied into contiguous memory and are under an eager size limit; finally the other is for returning a descriptor and memory for messages that need to be copied into contiguous memory and are larger than the eager size limit and less than or equal to the maximum size limit.

5.2.4 The Shutdown Phase

The shutdown phase is responsible for releasing resources at the end of an Open MPI job. There are two functions that are of importance: `btl_finalize()` and `component_close()`. The `btl_finalize()` function is very simple, it makes two iteration over all the endpoints created during the process selection phase. The first iteration shuts down the sending socket of each endpoint structure; a check is put in place to check that the socket was established in the first place. The next iteration shuts down the receiving socket of each endpoint structure and then releases the endpoint object. Provided Open MPI hasn't retained the object for itself, this will call the destructor and free the object from memory. After this the `xen` module is freed. The `component_close()` function is even simpler; it merely calls any destructors on objects (e.g. lists) created during the `component_open()` function. It is higher layers that allocate and free memory for the component itself.

5.2.5 Live Migration

Failover on live migration was an unfinished feature of the `xen` component. Theoretically it should be possible, but due to time constraints it was not properly investigated. The following is an explanation of problems found when performing a live migration during a running Open MPI job.

Given a scenario of two co-resident virtual machines running one Open MPI process each, there will be a `xen` endpoint established by each process to communicate with the other. A `xen` endpoint contains two XenSocket file descriptors, one for receiving and another for sending. It is the receiving XenSocket that owns the memory of the internal buffer whereas the sending XenSocket maps this memory into its address space and does not allocate any of its own. If one virtual machine

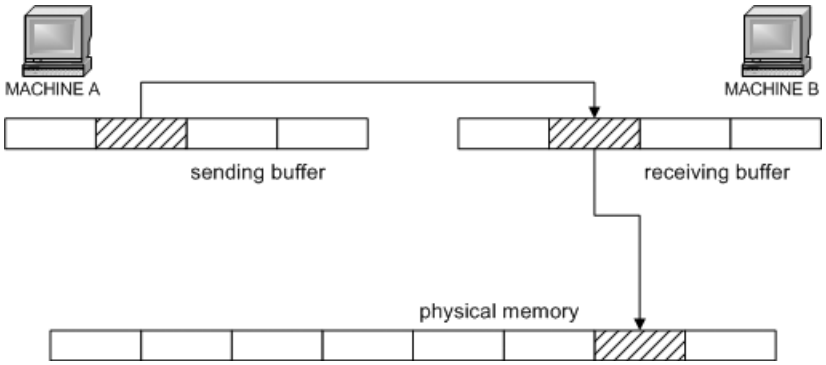

Figure 5.6: Both ends of a XenSocket in virtual memory

migrates to another physical machine, its entire memory is transferred too. This implies that the internal buffer of the receiving XenSocket is safely be transferred over to the new host so there is no risk of losing any unreceived data.

The problem lies in the sending XenSocket's behaviour which can vary depending on circumstance. Ideally, when the virtual machine hosting the matching receiving XenSocket migrates, the sending XenSocket should immediately break; this way the socket can return an error code that could be inform the `xen` *BTL* component of the problem. It was found using a looping unidirectional blocking send MPI program that if the machine performing the send operation migrates during the job, the sending XenSocket will break as expected; however, if the machine performing the receive operation migrates, the sending XenSocket will remain operational as if the memory is still mapped. This was unexpected behaviour.

The following is an attempt to explain what may be happening. Figure 5.6 shows virtual machine B, which actually owns the memory of allocated buffer for the XenSocket and also shows virtual machine A which doesn't own the memory, but instead maps it into its virtual address space. For clarity, virtual machine A's reference to the memory is seen going through virtual machine B's virtual memory address. Figure 5.7 shows the result of machine A emigrating to a new physical machine. Because the sender doesn't own the memory in the allocated buffer, it isn't transferred during the migration. This leaves the sender with an invalid memory reference which is exactly what is needed in order to report an error. The problem is shown in figure 5.8 where machine B migrates instead. The migration process makes a new a new copy of the allocated buffer in the new physical machine and machine B won't notice any difference; however, it appears that the memory in the original host isn't deallocated after the migration. This results in machine A getting a type of psuedo-ownership over the allocated buffer.

During the design stage of this project an assumption was made that both migration scenarios would lead to the sending XenSocket's buffer becoming invalid.

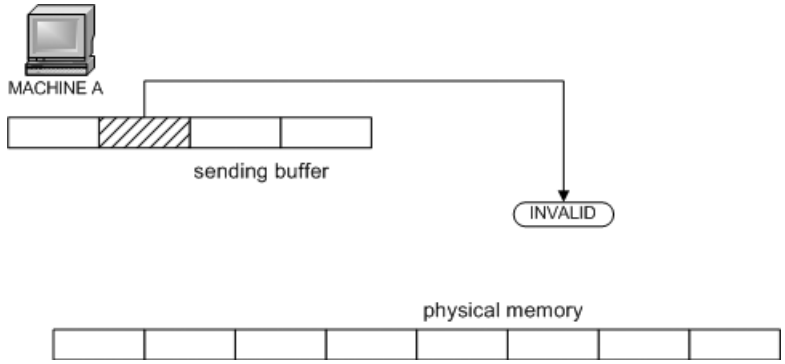

Figure 5.7: The sending machine emigrates

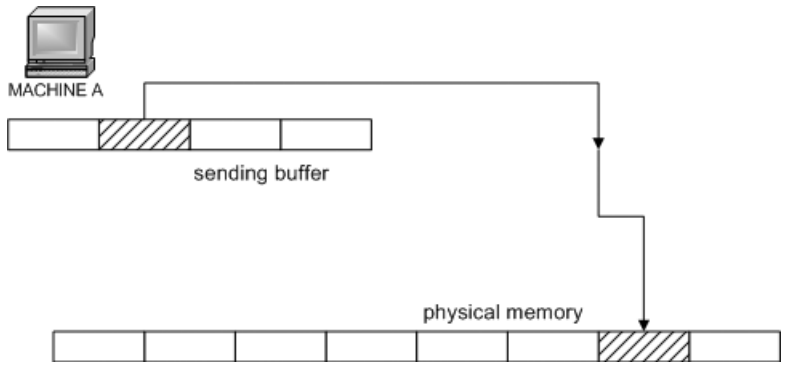

Figure 5.8: The receiving machine emigrates

This ultimately resulted in the live migration feature being incomplete. The problem could be resolved if there is a Xen construct that can identify if some mapped memory's original owner has migrated to another host. This way a check could be made at the end of a send operation which would return an error code appropriately. Given an error, it could be used to tell higher levels in Open MPI not to use the endpoint any more. Provided the job is run with the `tcp` component too, Open MPI will automatically use these endpoints instead.

In the duration of this project, no such feature in Xen was found. Other techniques could be used instead, for example using XenStore to check for migrations. However, this would incur a huge overhead which would probably cancel out any performance gains achieved in the first place.

6 Analysis

The following benchmarks were made using a simple MPI blocking ping-pong program. The machine was an AMD Athlon 64 X2 4200+ @ 2200 MHz; 1Ghz HyperTransport FSB; 1MB level-2 shared cache; 4GB of DDR2-800 RAM. All tests were carried out on a Xen 3.3 hypervisor between two domU virtual machines running a 2.6.27 Linux kernel. Each virtual machine was assigned one virtual processor that corresponded to one unique core of the physical processor (CPU affinity was enabled) and allocated 256MB of RAM. For any shared memory test an extra processor was given to the virtual machine. Latency/Bandwidth benchmarks were run 6 times taking the arithmetic mean of the results.

6.1 Latency & Bandwidth

Figure 6.1 shows a latency comparison between the tcp, xen and sm *BTL* components. Both axis are shown on a logarithmic scale for readability. It is immediately obvious that the xen component consistently outperforms the tcp component. For 1048576 bytes, the latency of the tcp component is 6972µs in contrast to the xen component which is 3453µs which is less than half the time. By the end the sm component outperforms the xen component with a latency of 1803µs which is roughly half the time of the xen component and one quarter the time of the tcp component. For smaller messages there is a greater margin of latency, for example 512 bytes takes 103µs using the tcp component, 15µs using the xen component and 24µs using the sm component. Therefore the xen component takes 15% of the tcp component's time, but also outperforms the sm component.

Figure 6.2 shows a bandwidth comparison between the tcp, xen and sm *BTL* components. The horizontal axis is shown on a logarithmic scale for readability. Once again it is clear that the xen component consistently outperforms the tcp component. For 1048576 bytes, the bandwidth of the tcp component is 301MBp/s in contrast with the xen component which is 608MBp/s, this is over double the throughput. For the same amount of bytes, the sm component has a bandwidth of 1163MBp/s which is roughly double the throughput of the xen component and four times the throughput of the tcp component. Therefore the bandwidth comparison is inversely proportional to the latency comparison. Similar to the latency comparison, for smaller messages the margin of bandwidth is greater. For 512 bytes, the tcp component has a throughput of 10MBp/s, the xen component has a throughput of 69MBp/s and the sm has a throughput of 104MBp/s. This shows that for certain message sizes, the bandwidth of the xen component is almost 10x greater than that of the tcp component.

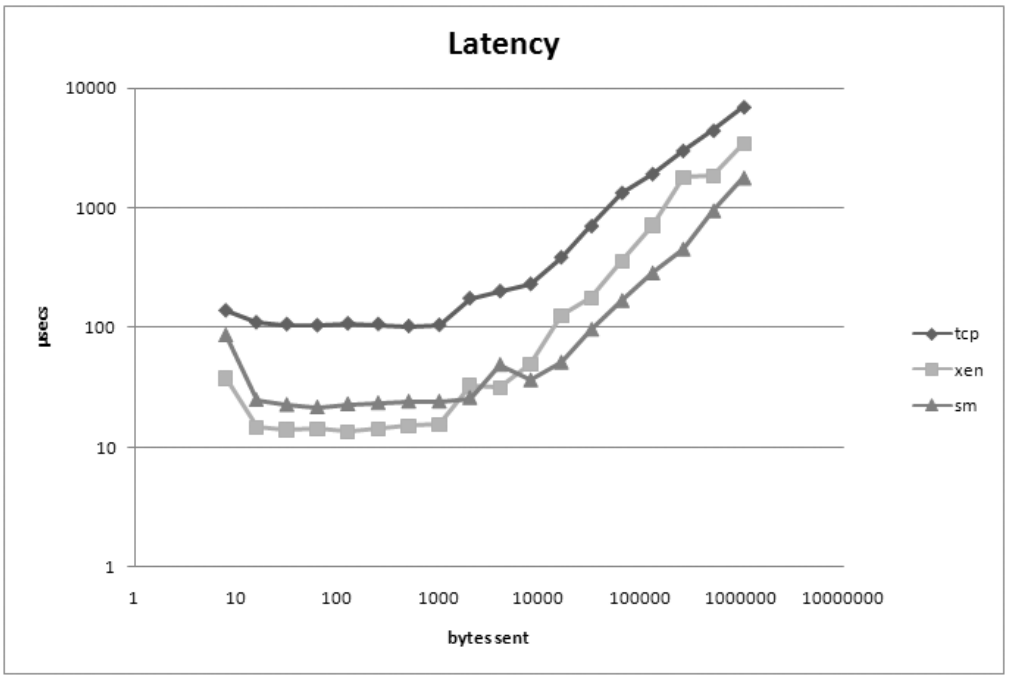

Figure 6.1: Latency of the tcp, xen & sm components

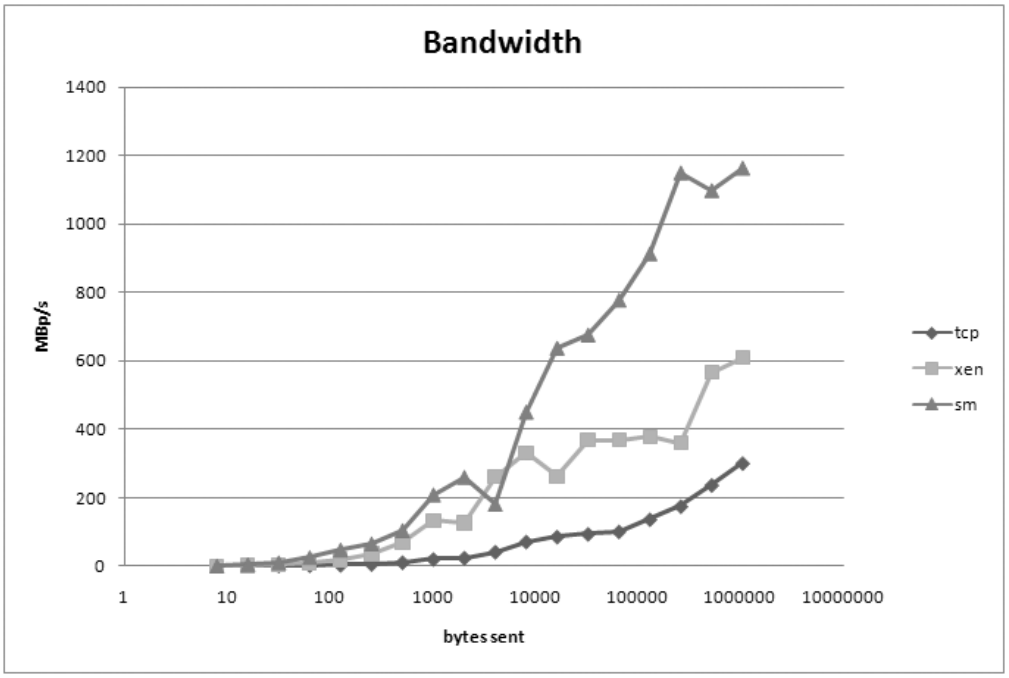

Figure 6.2: Bandwidth of the tcp, xen & sm components

6.2 CPU Usage

CPU activity on the virtual machines was measured using Xenoprof[19], a customised version of OProfile[20]. A finite looping unidirectional blocking send/receive program was used for these benchmarks.

Figure 6.3 shows the CPU activity on the guest virtual machines executing the Open MPI job. `ompi btl` refers to the activity in the `tcp` and `xen` *BTL* component libraries themselves; `ompi` refers to any other stimulus in the three Open MPI code sections. While the actual program and *BTL* components have roughly the same amount of CPU activity, it's clear that the `tcp` component has more interaction with the rest of Open MPI as well as the standard C library. There is a significant difference in kernel activity between both of components; the `xen` component has only 12% of the `tcp` component's kernel time. The `xen` component does have more activity in the Xen hypervisor which is due the granting and mapping of pages during the connection setup. In total, the `xen` component has over 5x less CPU activity than the `tcp` component making it comparatively light weight.

Figure 6.4 shows the CPU activity on dom0 during the execution of the Open MPI job run on two other guest virtual machines (i.e. dom0 did not run any Open MPI code itself). Stimulus from the `xen` component job is 60% of the `tcp` component job. It is interesting to note that there were only 12 samples taken from the software bridge which may suggest that the activity in the dom0 isn't as significant as outlined in section 3. Various unofficial sources revealed that networking had been improved since a lot of the initiatives described in section 3 took place, however the exact nature of what has been changed is not clear.

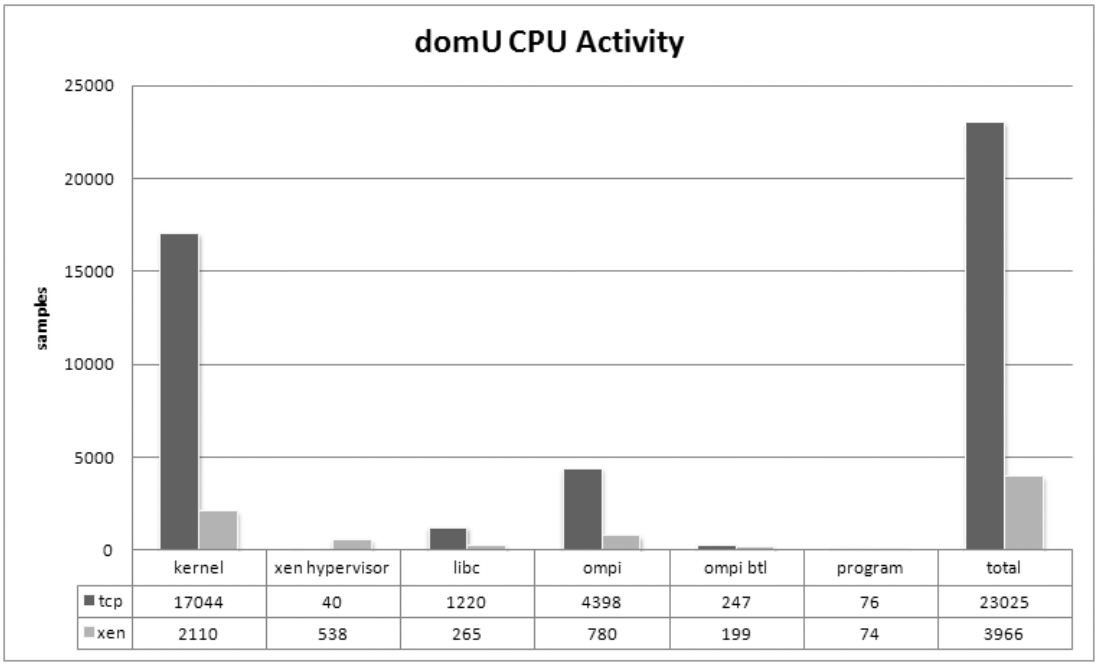

Figure 6.3: domU CPU activity of the tcp and xen components

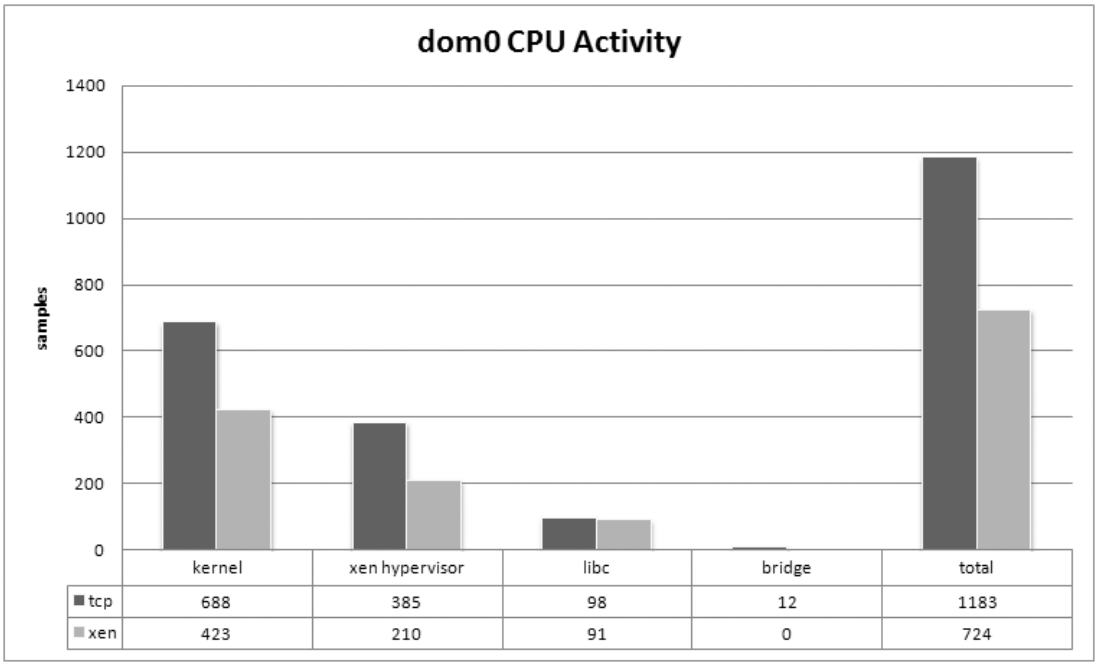

Figure 6.4: dom0 CPU activity of the tcp and xen components

6.3 The Alternative Design

Section 5 detailed that there were two designs of the xen *BTL* component. While the principle design was benchmarked against the tcp and sm components, we now compare the principle design with the alternative design: xenalt.

Figure 6.5 shows the latency of the xen and the xenalt components. The horizontal axis is shown at logarithmic scale for readability. The shape of the graphs are very similar, however there is some variation. The xen component takes slightly less time when it comes to shorter messages, but after 32768 bytes the xenalt component shows signs of improvement, for example 131072 bytes takes 717µs using the xen component and only 395µs using the xenalt component.

Figure 6.6 shows the bandwidth of the xen and the xenalt components. The horizontal axis is shown at logarithmic scale for readability. The graphs take a similar shape including the negative spike at 16384 bytes. After 32768 bytes, the xenalt starts performing better than the original design and there is a significant amount of bandwidth gained. By the time both components reach their peak at 1048576 bytes, the xen component has a throughput of 608MBp/s whereas the xenalt component has a throughput of 757MBp/s which is an improvement by a factor of 1.2x.

A reason this may be happening is because in the original xen design, the XenSocket atomically updated its internal buffer size for each `iovec` structure in the message being sent. There are usually three `iovec` structures in a message, one for the *BTL* header, one for the *PML* header and another for the message itself. This means the receiving process could poll the XenSocket just after one of these `iovec` structures had been transferred and proceed to receive whatever is in the XenSocket buffer. This would result in the receiving process making a kernel call just to receive a relatively small header leaving the rest of the message for the next polling cycle. In contrast, the alternative xenalt design modified its XenSocket to have an internal counter that is only incremented on the receipt of entire messages. Polling simply checks to see if this counter is above zero and proceeds to receive a single message over the space two system calls. While there are two system calls to receive a single message, they happen very close to each other instead of between an additional circuit of the Open MPI progress engine.

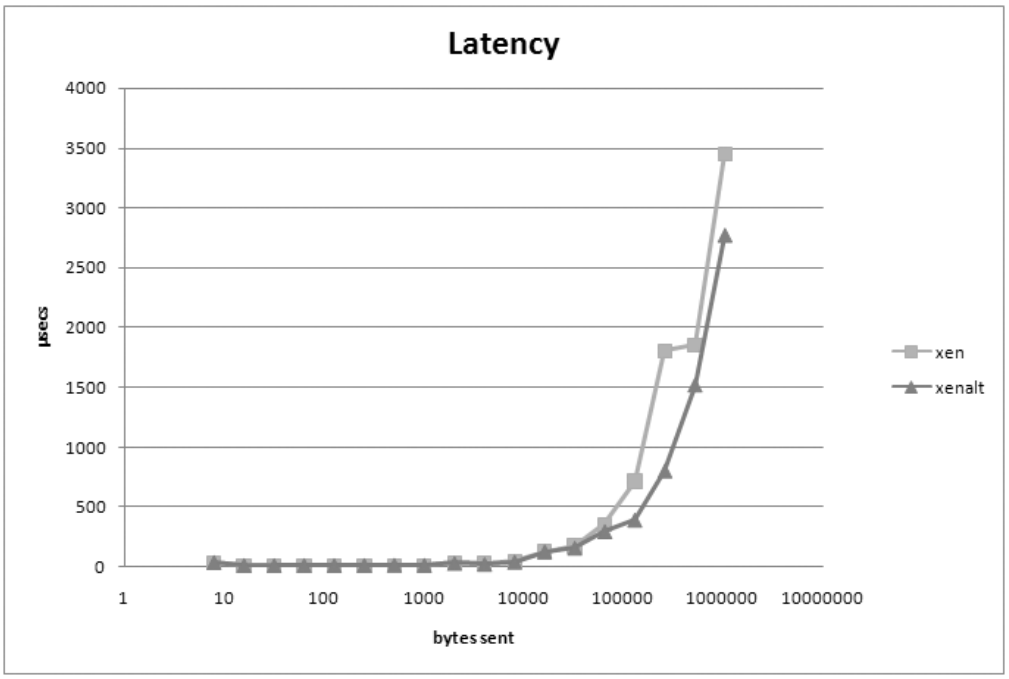

Figure 6.5: Latency comparison of alternative xen designs

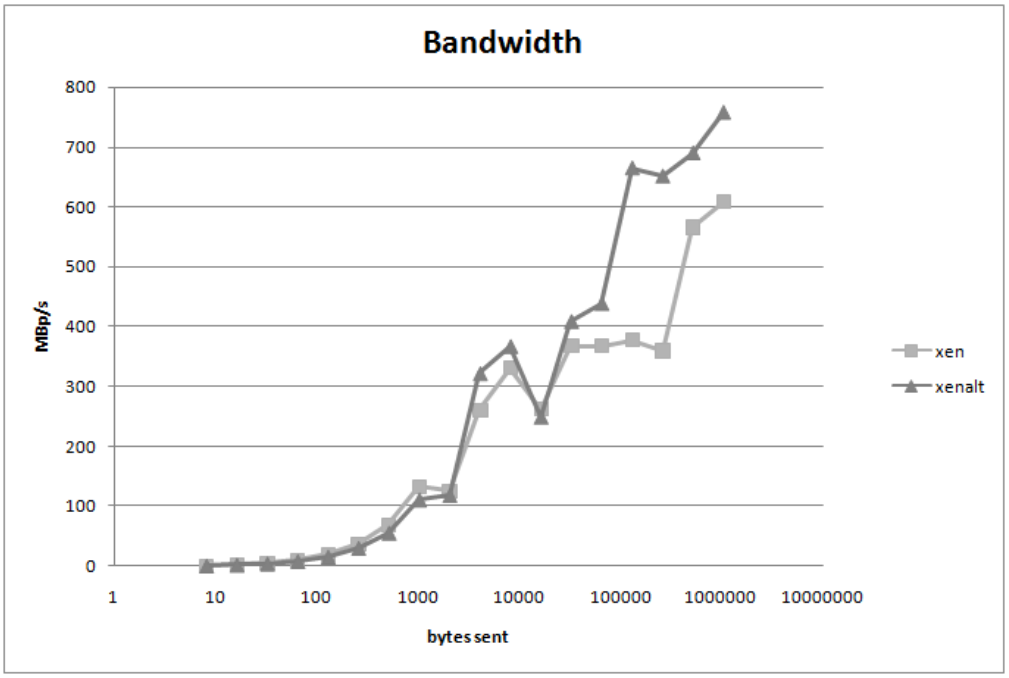

Figure 6.6: Bandwidth comparison of alternative xen designs

7 Summary

The following section looks at some of the problems encountered during the project's design and implementation and also concludes the project as a whole.

7.1 Problems Encountered

Open MPI is industry-ready software, so it's no surprise that it is both large and complicated. In theory, a component could be implemented by following a simple interface, but in reality the dependencies go right down to the core components of the framework. While the *BTL* component/module isnt too large, its API and expected behaviour remain quite undocumented. Most of the research for Open MPI was done by examining previously written *BTL* components and stepping through them with a debugger to observe their interaction with the rest of system.

From the beginning it had been decided to work from the most current stable release of Open MPI (version 1.2). At the time the team were preparing the next major version and it was decided not to use it as the version was still in beta stages. After the component had been implemented, there was a reoccurring but non-deterministic deadlock issue. It took some time to trace where it was happening, but it was eventually found to be caused by higher layers of the Open MPI framework. This was a known bug and fixing the problem meant upgrading to a newer release of Open MPI (version 1.3). The new version had fixed the problem but had also changed the API for the *BTL* framework. The problem promptly vanished after the xen component was upgraded to work with this new release.

In the original design of the xen *BTL* an alternative operation phase was implemented based on Linux real-time signals instead of polling. A signal would be sent to the receiving process whenever there was data pending in the XenSocket buffer. The reasoning behind this decision was that real-time signals could contain an additional piece of data that could be sent to the signal handler. In order to avoid the polling system completely, the file descriptor of the socket was passed with the signal. The problem with this was that a signal was generated for every message in the XenSocket buffer and the receiving process could become oversubscribed with signals too easily. It was found that signals were being lost, and while it is possible to increase the amount of real-time signals queued for a process, it became apparent that the design wasn't very scalable. The real-time signal design was abandoned in favour of polling which better suited the Open MPI model anyway.

The biggest problem came in the form of data being lost in the XenSocket buffers. This happened non-deterministically and was incredibly difficult to trace

and debug. It was eventually found that Xen was not guaranteeing atomicity of `atomic_*` kernel instructions in XenSocket. After some correspondence with the developers' mailing list, it was found that Xen will dynamically remove the `LOCK_PREFIX` from the standard `atomic.h` header when run on a virtual machine with a single virtual processor. Although both virtual machines had guest operating systems with `CONFIG_SMP` defined, all the locking assembly was getting filtered out. To fix this, a custom `atomic.h` was made replacing all references to `LOCK_PREFIX` with explicit assembly locking code.

7.2 Conclusion

In the course of this project, an efficient transport system for co-resident virtual machines running MPI jobs was created. The transport system outperformed the default networking model, having decreased its latency down to as much as 15% of the original time; increased its throughput by up to 1000% and reduced its CPU utilisation by more than a factor of 5. The transport system bypasses the native inter-VM communication system by making direct channels between virtual machines; this was shown to be both efficient and safe without breaking the virtual machine's security or isolation. The transport system works without any modification to the dom0, making it more favourable over alternative solutions described in section 3. The transport mechanism comes in the form of an easily installable Open MPI component which can be used without modification.

The additional goal of this project was to have *BTL* failover on live migration. This was not successful, however the nature of the problem preventing this from working was identified. With some more research into Xen's API a solution might be found. The problem itself is simple in nature and once resolved there should be nothing to prevent the `xen` module from using Open MPI's failover mechanism.

Aside from the component implementation, there may be some valuable information in the report itself. When starting the project, it was found that Open MPI was quite an overwhelming piece of software. It was hard to find details on the functionality and behaviour of many of its features outside of the usual function annotations. This report was written with a meticulous amount of detail in the hope that it might benefit programmers new to Open MPI.

This project successfully demonstrated that some of the overhead incurred from virtualization can be eradicated. Although the `xen` component didn't perform as well as the inter-process `sm` component, there was definitely a significant improvement over the `tcp` component. This may give some incentive to the HPC community to consider adopting virtualized solutions. The points outlined in section 1 show there are great benefits to virtualization and although there is, and will always be, some overhead involved, using this efficient transport system can reduce it significantly.

8 Bibliography

[1] David Chisnall (2007), *The Definitive Guide to the Xen Hypervisor*, Prentice Hall

[2] Jeanna N. Matthews, Eli M. Dow; Todd Deshane, Wenjin Hu, Jeremy Bongio, Patrick F. Wilbur and Brendan Johnson (2008), *Running Xen: A Hands-On Guide to the Art of Virtualization*, Prentice Hall

[3] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman (2005), *Linux Device Drivers, Third Edition*, O'Reilly

[4] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall (2004), *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*

[5] Richard L. Graham, Timothy S. Woodall and Jeffrey M. Squyres (2005), *Open MPI: A Flexible High Performance MPI*

[6] Brian Barrett, George Bosilca, Rich Graham, Galen Shipman, Tim Woodall and Jeff Squyres (2006), *Open MPI Developer's Workshop*, available at `http://www.open-mpi.org/papers/workshop-2006/`

[7] Mark F. Mergen, Volkmar Uhlig, Orran Krieger and Jimi Xenidis (2006), *Virtualization for High-Performance Computing*

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield (2003), *Xen and the Art of Virtualization*

[9] Padma Apparao, Srihari Makineni and Don Newell (2006), *Characterization of network processing overheads in Xen*

[10] Hyun-Sup Shin, Kang-Ho Kim, Chei-Yol Kim and Sung-In Jung (2007), *The new approach for inter-communication between guest domains on Virtual Machine Monitor*

[11] Kangho Kim, Cheiyol, Kim Sung-In Jung, Hyun-Sup Shin and Jin-Soo Kim (2008), *Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen*

[12] Jian Wang, Kwame-Lante Wright and Kartik Gopalan (2008), *XenLoop: A Transparent High Performance Inter-VM Network Loopback*

[13] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi and John Linwood Griffin (2007), *XenSocket: A High-Throughput Interdomain Transport for Virtual Machines*

[14] François Diakhaté, Marc Perache, Raymond Namyst and Hervé Jourdren (2008), *Efficient Shared Memory Message Passing for Inter-VM communications*

[15] Message Passing Interface Forum (2008), *MPI: A Message-Passing Interface Standard Version 1.3*

[16] Message Passing Interface Forum (2008), *MPI: A Message-Passing Interface Standard Version 2.1*

[17] Dan Kegel (2006), *The C10K problem - Why can't Johnny serve 10000 clients?*, available at `http://www.kegel.com/c10k.html`

[18] Niels Provos, *libevent - an event notification library*, available at `http://www.monkey.org/~provos/libevent/`

[19] *Xenoprof - System-wide profiler for Xen VM*, available at `http://xenoprof.sourceforge.net/`

[20] *OProfile - A System Profiler for Linux*, available at `http://oprofile.sourceforge.net/`

A Source Code

A.1 CD-ROM Contents

open-mpi-1.3.2.tgz contains a complete copy of Open MPI 1.3.2-stable.

xen.tgz contains the code for the `xen` *BTL* component.

xensocket.tgz contains the code for the updated custom XenSocket.

A.2 Installing Software

To use the `xen` *BTL* component, first install the customised XenSocket. Untar `xensocket.tgz` and as a root user, issue the following commands.

```
# make
# insmod xensocket.ko
```

Listing A.1: Building and installing XenSocket

After XenSocket is installed, untar `open-mpi-1.3.2.tgz` and then untar `xen.tgz` into *open-mpi-1.3.2/ompi/mca/btl/* and issue the following commands. The `make install` command may require root privileges.

```
# ./autogen.sh
# mkdir build
# cd build
# ../configure --with-platform=optimized
# make
# make install
```

Listing A.2: Building and installing Open MPI with the `xen` component