# vNUMA: Virtual Shared-Memory Multiprocessors


A thesis submitted to the School of Computer Science and Engineering at The University of New South Wales in fulfilment of the requirements for the degree of Doctor of Philosophy.

Matthew Chapman

2008

## Originality statement

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

## Copyright statement



## Authenticity statement

I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.

# Abstract

Shared memory systems, such as SMP and ccNUMA topologies, simplify programming and administration. On the other hand, systems without hardware support for shared memory, such as clusters of commodity workstations, are commonly used due to cost and flexibility considerations.

In this thesis, virtualisation is proposed as a technique that can bridge the gap between these architectures. The resulting system, vNUMA, is a hypervisor with a unique feature: it provides the illusion of shared memory across separate nodes on a fast network. This allows a cluster of workstations to be transformed into a single shared memory multiprocessor, supporting existing operating systems and applications. Such an approach could also have applications for emerging highly-parallel architectures, allowing a shared memory programming model to be retained while reducing hardware complexity.

To build such a system, it is necessary to meld both a high-performance hypervisor and a high-performance distributed shared memory (DSM) system. This thesis addresses the challenges inherent in both of these tasks. First, designing an efficient hypervisor layer is considered; since vNUMA is implemented on the Itanium processor architecture, this is with particular reference to Itanium processor virtualisation. Then, novel DSM protocols are developed that allow SMP consistency models to be reproduced while providing better performance than a simple atomically-consistent DSM system. Finally, the system is evaluated, proving that it can provide good performance and compelling advantages for a variety of applications.

# Publications

Portions of this work have been published in the following articles:

- **Implementing transparent shared memory on clusters using virtual machines**
  Matthew Chapman and Gernot Heiser
  Proceedings of the 2005 USENIX Technical Conference, Anaheim, CA, USA, April 2005
  *Introduces vNUMA concept and presents an early version of the system*

- **vNUMA: A Virtual Shared-Memory Multiprocessor**
  Matthew Chapman and Gernot Heiser
  To appear in Proceedings of the 2009 USENIX Technical Conference
  *Summarises the work in this thesis*

- **Itanium: a system implementor's tale**
  Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang and Gernot Heiser
  Proceedings of the 2005 USENIX Technical Conference, Anaheim, CA, USA, April 2005 *(Best Student Paper Award)*
  *Describes Itanium virtualisation challenges — Chapter 3 in this thesis*

- **Pre-virtualization: soft layering for virtual machines**
  Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie and Gernot Heiser
  Proceedings of the 13th Asia-Pacific Computer Systems Architecture Conference, August 2008 *(Best Paper Award)*
  *Describes pre-virtualisation — Chapter 4 in this thesis*

- **[para]virtualisation without pain**
  Peter Chubb, Matthew Chapman and Myrto Zehnder
  Proceedings of the 8th Linux.Conf.Au, January 2007
  *Describes pre-virtualisation — Chapter 4 in this thesis*

- **Pre-virtualization: slashing the cost of virtualization**
  Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie and Gernot Heiser
  Technical Report PA005520, National ICT Australia, October 2005
  *Describes pre-virtualisation — Chapter 4 in this thesis*

# Acknowledgements

This thesis would not have been completed without the encouragement and assistance of a number of people.

I am indebted to my supervisor, Professor Gernot Heiser, who stood behind me and motivated me to bring this work to fruition, as well as providing feedback on drafts of this thesis and the preceding papers. Similarly, I appreciate the valuable feedback and helpful advice of my co-supervisor, Dr Peter Chubb, as well as Dr Felix Rauch.

I would also like to take the opportunity to thank my fellow research students, particularly Ian Wienand, Charles Gray and Daniel Potts, who were always prepared to listen to my problems and discuss my solutions — despite, or perhaps due to, having theses of their own to worry about.

Finally, the pre-virtualisation idea that is used to minimise virtualisation overheads arose out of collaborative work with colleagues at the University of Karlsruhe, particularly Joshua LeVasseur and Volkmar Uhlig. However, the specific embodiment for the Itanium architecture is unique to this thesis.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Architectures for multiprocessor systems can be classified into two groups: those with coherent shared memory, such as SMP and ccNUMA systems, and those without, such as networks of commodity workstations connected by Ethernet.

Shared memory systems provide a simple programming model compatible with a large base of existing applications and operating systems. The majority of existing operating systems support such architectures; it is a relatively easy task to support them, even if not with optimal performance. Further, they naturally lend themselves to *single system image* (SSI) systems, where there is a single instance of the operating system with a single resource namespace. The appearance of a single system makes life simple for users and administrators. The single resource namespace means that computation can transparently migrate between processors to balance load, without fear that references to resources will become invalid.

Despite this, there has been a trend towards utilising networks of commodity workstations. Due to economies of scale, commodity hardware is far more cost-effective than large shared memory systems. It also provides benefits in terms of easy extensibility and reconfigurability of a cluster. However, most existing operating systems were not designed with such a cluster environment in mind, and cannot provide a single system image across multiple nodes in the absence of shared memory. Applications that were previously designed for shared memory systems need to be redesigned to explicitly communicate via the network. Users typically need to be aware of the nodes available and the resources available on those nodes. Administrators need to maintain the configuration of the separate

nodes.

There are many previous attempts to bridge this gap by emulating certain desirable features of a shared memory system on a cluster. For example, distributed shared memory libraries such as Treadmarks [50] can provide a limited illusion of shared memory to the programmer, assuming that the programmer uses the primitives provided by the library. Other projects have attempted to retrofit support for cluster-wide process scheduling and migration into existing operating systems; for Linux such systems include MOSIX [6], openSSI [89] and Kerrighed [71]. However, such efforts require extensive and intrusive operating system changes, which are difficult to keep up to date with the fast pace of operating system development. Of course, it is also possible to build operating systems that are specifically designed to provide transparency in networked environments, but unfortunately such novel operating systems rarely gain traction due to the inertia of legacy code.

Recently virtualisation has re-emerged into the limelight, and is becoming a standard part of the data center. Virtualisation inserts an extra layer into the software stack, decoupling an operating system from the underlying hardware. The most common use of virtualisation is to allow multiple operating system instances to share a single computer. However, it has been recognised that virtualisation also has other interesting applications, such as enabling legacy operating systems to support new hardware architectures [8] or transparently migrating an operating system instance from one computer to another [15].

This thesis explores using virtualisation to bridge the gap between shared memory systems and workstation clusters. In the system described in this thesis — dubbed vNUMA, an acronym for virtual NUMA — a virtual shared memory multiprocessor is built from a cluster of workstations. Users gain all of the advantages of a single-system-image multiprocessor, such as a single view of resources and transparent process scheduling, while being able to use existing operating systems and applications.

An example of the architecture of a vNUMA system is shown in Figure 1.1. At a physical level, the system consists of a cluster of commodity workstations connected by Gigabit Ethernet. An instance of the vNUMA hypervisor runs on each workstation and is responsible for controlling physical resources such as

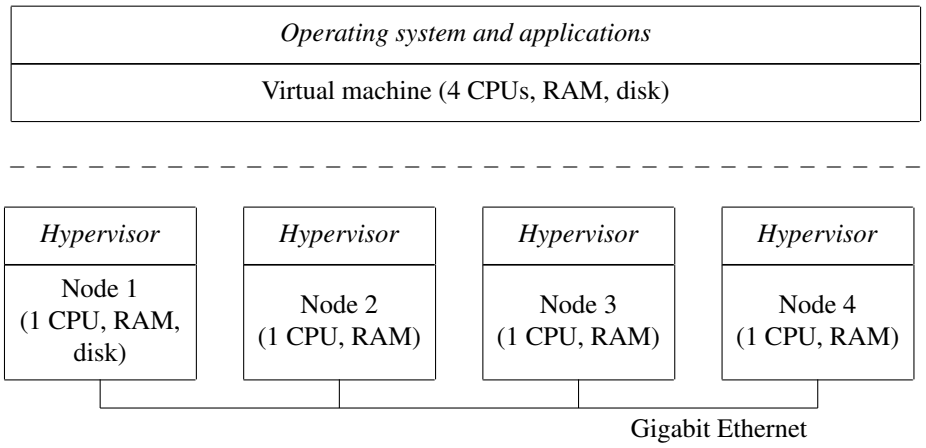



**Figure 1.1:** Example vNUMA system

processors, memory and devices (there is no host operating system). Initially, a virtual machine is started from one of the nodes of the cluster, and boots an operating system such as Linux. The hypervisors then co-operate to extend this virtual machine across all of the cluster's resources: transparently mapping virtual processors to real processors in the cluster, and providing shared memory using software techniques. In this way, a single operating system instance can be scaled 'outside the box', utilising the computing resources of more than one node. In this thesis, feasibility of this approach is investigated for a small cluster of up to eight workstations.

Since this thesis was commenced, two other systems have emerged which apply similar ideas to vNUMA: Virtual Iron's VFe hypervisor [88] and Kenji Kaneda's Virtual Multiprocessor from the University of Tokyo [49]. These should be considered to have been developed independently, although the vNUMA work described here was the first to be published. While these systems both demonstrate combining virtualisation with distributed shared memory, both are limited in scope and performance, and do not address many of the challenges that this thesis addresses. In particular, both use simpler virtualisation schemes and distributed shared memory protocols, resulting in severe performance limitations, especially in the case of Virtual Multiprocessor. Virtual Iron attempted to address some of these performance issues by using high-end hardware, such as InfiniBand rather

than Gigabit Ethernet. However, this greatly increases the cost of such a system, and limits the target market. Virtual Iron has since abandoned the product for commercial reasons, which largely seems to stem from its dependence on such high-end hardware[1]. vNUMA, in contrast, demonstrates how novel techniques can achieve good performance on commodity hardware.

Part I of this thesis analyses the virtualisability of the Itanium processor architecture upon which vNUMA is built, and presents an approach for efficient virtualisation. Part II investigates the challenges involved in implementing a distributed shared memory system within this virtualisation layer, that can satisfy the demands of an operating system and unmodified applications while simultaneously providing good performance. It describes a distributed shared memory protocol that can meet these challenges. Part III evaluates the design of the system, presenting results for a variety of benchmarks.

[1] According to Virtual Iron's John Thibault, "Trying to sell InfiniBand into enterprise datacenters was, to say the least, a real challenge. We were spending more time selling InfiniBand than our own product." [83]

# Part I

# Itanium virtualisation

The vNUMA system is fundamentally a virtual machine which can cross the boundaries of physical computers. The first task, then, is to build the virtual machine infrastructure, and this will lay the groundwork for the second part of the thesis which investigates how it can be distributed across multiple physical computers.

While the ideas behind vNUMA can be applied to most processor architectures, there are significant practical differences between architectures, and it is necessary to choose one as a focus for this thesis and particularly the experimental prototype. The Itanium processor architecture was chosen for a number of reasons. Firstly, at the time this thesis was commenced, there was no other Itanium virtualisation solution or analysis of the virtualisability of the Itanium architecture. Intel was poising Itanium to become the next major server architecture, and so this was an important gap that needed to be filled. As of 2008, some would say that Intel's vision did not succeed, and that the 'Itanic' has been relegated to certain niche parts of the market. Nevertheless it is still widely used in scientific clusters and enterprise-class database servers, which are some of the workloads that vNUMA is designed for. The Itanium architecture also has many innovative features, such as its register stack engine and relaxed memory consistency model. These features provide opportunities for innovation, and are representative of a more modern architecture design than the ubiquitous x86 architecture, which is the result of 30 years of haphazard evolution. The virtualisation challenges associated with x86 are complex, widely known, and littered with patents.

This chapter will provide an overview of virtualisation, the issues involved with virtualising the Itanium architecture, and the design of the vNUMA virtualisation layer.

# Chapter 2

# Virtualisation

Software applications rely on an operating system to provide basic services such as input and output. Traditionally the operating system has been part of the lowest layer of software on a computer system, with full control of the system hardware. In such a design it is not possible to have more than one operating system on a single computer. However, the complexity and variety of modern operating systems have led to them being considered almost part of the application layer; developing a given application is intricately connected to the underlying operating system and its configuration. An operating system, its configuration and a set of applications that execute atop it can collectively be referred to as an operating system *instance*.

Virtualisation inserts another thin layer of software, known as a *virtual machine monitor* (VMM) or *hypervisor*, underneath the operating system. Instead of having direct control of the real computer hardware, the operating system now controls the resources of an imaginary computer known as a *virtual machine*; the virtual machine monitor controls how those resources are mapped to real hardware. The virtualisation layer enables a variety of scenarios for managing operating system instances.

One of the most common applications of virtualisation is server consolidation. The pace of progress in computer hardware is such that many servers are increasingly underutilised. Server consolidation allows multiple operating system instances, that might previously have been housed on separate servers, to be consolidated onto a single server. This improves resource utilisation and reduces

the number of physical computers required, resulting in cost, power and manageability advantages. Compared to simply combining applications onto a single operating system instance, virtualisation can provide stronger isolation between the applications, improving security. Virtualisation also allows consolidating workloads which require different operating systems, different versions of an operating system, or incompatible operating system configurations.

Furthermore, virtualisation enables easy migration of operating system instances between physical servers. In fact, a number of modern virtual machine monitors support *live* migration, where virtual machines can be migrated while still running, with minimal downtime [15]. This is useful for load balancing; entire virtual machines can be moved around to optimise resource utilisation.

Virtualisation permits the complete state of a virtual machine to be captured at any point in time. This allows simple cloning of servers, restoring servers to a known-good state, capturing snapshots for post-mortem analysis, making consistent backups of a live system with no downtime, and many other possible scenarios. One particularly novel application is the time-travelling virtual machine [53], which can be rewound to any point in time to aid in debugging. The virtual machine is first rewound to the last snapshot before the desired point, and then run forwards while deterministically replaying external events.

The encapsulation of operating system instances provided by virtualisation has also led to the emergence of *virtual appliances* as a mechanism for software deployment. Virtual appliances are packaged snapshots of virtual machines that have been pre-configured for a particular task. For example, a user may be able to download a web server appliance which contains not only the web server application, but an entire virtual machine which has been appropriately configured for that purpose. This is analogous to purchasing a 'turnkey' web server appliance, but in this case the hardware is virtual and consolidation is possible.

Finally, virtualisation can allow legacy software systems to execute on radically new hardware architectures, without imposing this compatibility burden on the hardware; the transformation between the two architectures is left to the virtual machine monitor. For example, MagiXen [11] allows operating systems designed for the ubiquitous x86 architecture to execute on the newer Itanium architecture. Disco [8] carves a NUMA system into multiple virtual SMP nodes

for the benefit of existing operating systems that may not support a NUMA architecture. vNUMA — the subject of this thesis — combines multiple separate nodes into a single virtual NUMA system, allowing a single operating system instance to span multiple physical servers.

## 2.1 Full virtualisation

Virtual machine monitors can be described in terms of two basic design philosophies: *full virtualisation* and *para-virtualisation*. In a VMM which implements full virtualisation, the interposition between the operating system and hardware is done transparently, such that the operating system does not need to be modified. In other words, the interface between the operating system and virtual machine monitor is the same as the original interface between the operating system and hardware.

This is made possible by the concept of privilege levels, which exist in practically all general-purpose processor architectures. At the least, processors have a privileged mode and an unprivileged mode. The privileged mode confers unfettered control of all aspects of processor operation, while the unprivileged mode is a carefully controlled sandbox; whenever an unprivileged program needs to reach beyond its sandbox, it must invoke the privileged layer. Traditionally the entity that runs at the privileged layer is the operating system. In a virtualised system, the operating system is relegated to the unprivileged mode together with other applications. When the operating system attempts to perform a privileged function, it is at first disallowed; the processor instead notifies the virtual machine monitor running in the privileged layer. The VMM performs any necessary effects, depending on the virtualisation scenario that is desired, and then makes it seem to the operating system that the operation succeeded normally. This interposition can often be made transparent to the operating system.

One of the earliest applications of this concept was IBM's CP/CMS system [17]. CP/CMS was developed in the 1960s as IBM's answer to the Multics time-sharing system. Instead of designing a completely new multi-user operating system, CP (Control Program) was a thin virtual machine monitor which multiplexed the resources of the physical machine. This allowed each user to start their own

instance of the CMS (Conversational Monitor System) operating system. The successors of CP/CMS, VM/370 and z/VM, are still in use today.

The same ideas were leveraged in the 1990s for a somewhat different purpose, in the form of the Disco system [8]. Disco was a virtual machine monitor for the MIPS processor architecture, specifically designed for the Stanford FLASH multiprocessor, a large experimental shared memory system. Upon the FLASH machine, Disco provided the abstraction of a cluster of SMP machines; each of these virtual SMP machines could execute a commodity MIPS operating system such as SGI IRIX. This circumvented the engineering effort that would be required to enhance such an operating system to run efficiently on the large multiprocessor. The subsequent Cellular Disco system [38] extended Disco to provide improved support for partitioning and resource management.

A number of the Disco researchers also went on to found VMware, Inc., which brought virtualisation into the limelight by applying it to the ubiquitous IA-32 processor architecture. IA-32 does not fulfil the necessary requirements for it to be directly virtualised via the trap-and-emulate mechanism described above (also known as *native* or *classic* virtualisation). As elucidated by Popek and Goldberg [79], the key requirement is all *sensitive* instructions — those that affect or depend on the current privilege level or the state of privileged resources in the system — are privileged. This is not the case for IA-32, which contains some unprivileged sensitive instructions; these can behave in subtly different ways in a lower privilege level without raising an exception [57]. VMware's breakthrough was to develop a relatively efficient method to translate privileged operating system code, at runtime, into code that has similar semantics but executes correctly in the lower privilege mode [1]. Microsoft Virtual PC and Microsoft Virtual Server, originally acquired from Connectix, also utilise comparable dynamic translation techniques. Making this dynamic translation fast and transparent to the operating system is no easy task, however, and this technology is closely guarded.

Since these products have taken hold in the market, processor designers have taken note. The largest manufacturers of IA-32 processors, Intel and AMD, have recently introduced virtualisation extensions to the IA-32 architecture, adding a new mode in which all of the problematic instructions can be intercepted [48, 2]. These extensions have since been leveraged by both the VMware and

Microsoft solutions, as well as those from other industry players. Intel has also added similar extensions to the Itanium processor architecture, which has similar problems to IA-32, as will be described in this thesis. On the other hand, these virtualisation extensions do not solve all of the challenges associated with virtualisation, particularly when good performance is required [1].

## 2.2 Para-virtualisation

Full virtualisation imposes significant limitations on a virtual machine monitor design, since it must accurately reproduce a hardware interface. If the architecture is not natively virtualisable and dynamic translation is needed, this introduces considerable complexity and runtime overhead. Even if the architecture lends itself to native virtualisation, the nature of such virtualisation is such that each privileged instruction causes an exception to the virtual machine monitor. On modern pipelined processors, these frequent exceptions result in a high overhead (as will be seen in Chapter 12).

For these reasons there has been a trend towards para-virtualisation, which adapts the interface to the virtual machine monitor for greater simplicity, performance and functionality. For instance, a para-virtualised version of an architecture might define that the operating system must not rely on the behaviour of certain instructions. Direct *hypervisor calls* may be introduced to streamline certain operations such as context switching or access to devices, rather than relying on emulation of privileged instructions. The downside of para-virtualisation is that an operating system must be explicitly modified to use the new interface, with the associated engineering overhead of maintaining a separate body of code that is not used on real hardware platforms.

The para-virtualisation technique is not new, in fact it was already applied *ante litteram* in the CP/CMS system. While CMS was originally designed so that it could execute on real hardware, very soon new functionality was introduced that specifically relied on CP features; for instance, efficiency gains were made by replacing the device accesses in CMS with direct calls to CP. This involved using the `DIAGNOSE` instruction, normally a reserved instruction that operating systems should not use, as a mechanism for invoking hypervisor calls [87].

More recently, this idea was taken further by the Denali project [90]. Denali sought to build a type of hypervisor (they called it an *isolation kernel*) that could multiplex thousands of lightweight virtual machines. In order to provide such scalability, far beyond the capabilities of conventional hypervisors, the overheads involved in virtualisation had to be reduced. Para-virtualisation was the answer. Rather than attempting to faithfully emulate the complicated privileged architecture of IA-32 processors, Denali exported a simpler set of abstractions. In this way it is also similar to a microkernel; in fact there is little practical difference between a hypervisor and a microkernel, except that the choice of basic primitives for a hypervisor tends to be biased towards the requirements of existing operating systems, whereas the design of microkernels is primarily guided by the desire for a minimal elegant interface. Indeed para-virtualisation methods can also be applied to allow existing operating systems to run on top of a microkernel, where the microkernel acts as a hypervisor. For example, the Wombat system [59] allows Linux to run atop the L4 microkernel.

Para-virtualisation has gained greatest traction in the form of Xen, which has risen to become one of the leading enterprise virtualisation platforms (alongside VMware's and Microsoft's offerings). Xen was originally developed in the context of Linux on the IA-32 architecture; it relies on major operating system changes on that platform, with most privileged functionality being achieved via hypervisor calls rather than privileged instructions. However, the extent of these changes has hampered the adoption of these modifications into the mainstream Linux source code, and a great deal of engineering effort is required to keep them up-to-date in the face of a fast-changing operating system.

The Itanium port of Xen, however, does not require such significant operating system changes. The code is originally derived from the vBlades hypervisor [63] and uses a technique that has been dubbed *optimised para-virtualisation*. The idea behind optimised para-virtualisation is to first implement a hypervisor that is capable of full virtualisation (or close to it), making minimal modifications to the operating system. Having done this, one can do detailed profiling to determine which code paths would benefit most from para-virtualisation, and so achieve a balance between performance and engineering effort.

## 2.3 Pre-virtualisation

To address the performance cost associated with full virtualisation and the engineering cost associated with para-virtualisation, LeVasseur et al (including the present author) recently introduced the idea of pre-virtualisation [60]. Pre-virtualisation is an automated form of para-virtualisation that can be used when the source code for the operating system kernel is available. Instead of requiring a programmer to manually modify the source code, it is transformed during the build process, by using a special toolchain that replaces certain instructions. The simplest option is simply to map each problematic instruction into an implementation tailored for a particular hypervisor (a simple version of such a technique was also used in the LilyVM virtual machine [25]). A more sophisticated possibility is to transform the code into an intermediate form that still executes correctly on real hardware but allows the problematic instructions to be easily identified and replaced at runtime.

The vNUMA hypervisor described in this thesis uses pre-virtualisation. However, pre-virtualisation was originally conceived with reference to the x86 architecture, whereas vNUMA is an Itanium virtual machine monitor; applying these techniques to the Itanium architecture is not trivial. Chapter 3 will describe general challenges associated with Itanium virtualisation, while Chapter 4 will specifically address applying pre-virtualisation to the Itanium architecture.

## 2.4 Type I vs Type II

Para-virtualisation and pre-virtualisation can also be applied to allow an operating system to run directly on top of another operating system. User Mode Linux [22] is a para-virtualised port of Linux to Linux, whereby Linux is considered to be another target architecture; hardware abstractions such as virtual address spaces are implemented in terms of Linux processes and APIs. Clearly this requires significant porting and maintenance effort, even greater than those changes required by Xen. Linux-on-Linux, which will be described in Chapter 5, is a less intrusive approach using pre-virtualisation, requiring minimal changes to the two operating systems.

It is useful to define some terminology in this context. User Mode Linux and Linux-on-Linux are examples of what is known as *Type II* [36] or *userspace* virtual machine monitors, in that they depend on an underlying operating system. That operating system is known as the *host* operating system, whereas the operating system inside the virtual machine is known as the *guest* operating system. In contrast, a *Type I* or *standalone* virtual machine monitor executes on bare hardware and does not utilise a host operating system. It is also possible to construct a *hybrid* virtual machine monitor, such as VMware Workstation and Microsoft Virtual PC, which utilise a host operating system for certain services but steal the processor from the operating system while the VMM is executing.

# Chapter 3

# Itanium challenges

This hypervisor described in this thesis has initially been designed for the Itanium processor architecture [47]. Itanium is a modern 64-bit processor architecture that was developed jointly by HP and Intel in the late 1990s. While its instruction set shares many features with HP's earlier PA-RISC architecture, Itanium has a number of novel features that set it apart in its own class.

Unlike most processor architectures, the Itanium architecture explicitly represents instruction-level parallelism in the instruction stream. This is described as *explicitly-parallel instruction-set computing* (EPIC). Instructions are arranged in bundles destined for specific execution units, and delineated into instruction groups that specify which instructions can be executed in parallel. This results in a simpler hardware design — the processor does not need to dynamically re-order instructions — but shifts the burden of complexity to compilers and low-level software.

Similarly, other processor architectures provide relatively small register sets; high-performance processor implementations then dynamically remap these onto a larger number of physical registers. Itanium instead provides a large architectural register set, eliminating the need for dynamic remapping. To aid software in the management of this large number of registers, the Itanium architecture provides a register windowing feature, similar to but more flexible than that provided by the SPARC architecture [84]. In addition to a set of 32 fixed registers, each procedure can allocate a variable-sized window of up to 96 registers. This window is divided into input, local and output portions; upon a procedure call,

the window is shifted so that the output portion becomes the input portion for the callee, and the caller's local registers are safely hidden. Even though the physical number of processor registers is limited, the Itanium architecture provides a *register stack engine* (RSE) which transparently saves and restores registers in the hidden portion, providing the illusion of an infinite continuum of registers.

This chapter will present the challenges involved in virtualisation of the Itanium architecture, and how they can be addressed. When the Itanium architecture was first developed, virtualisation was not seen as a design goal. As a result the Itanium architecture, like IA-32, is not natively virtualisable in the classic way. When an Itanium operating system is demoted to a lower privilege level — the first step towards virtualisation — a number of issues arise that compromise the ability of the hypervisor to provide a faithfully virtualised environment. These issues will be explored in the following sections.

It should be noted that a number of these problems were also encountered by the authors of vBlades [63], another virtual machine monitor for the Itanium architecture that was developed independently from vNUMA. Except where noted, vBlades employs very similar workarounds to vNUMA. The vBlades code also later became the foundation for Xen/ia64, the Itanium port of the Xen virtual machine monitor [29].

Most recently, Intel has added a set of extensions to the Itanium architecture, codenamed Silvervale but officially known as Virtualisation Technology for Itanium (VT-i) [47]. The first implementation of VT-i was released with the Dual-Core Itanium 2 ('Montecito') processor in 2006. VT-i addresses some but not all of Itanium's virtualisability issues. Specifically, it provides a special processor mode that alleviates the problems described in Sections 3.1, 3.2 and 3.3. While this brings Itanium closer to being natively virtualisable, this is not a panacea; as previously mentioned, para-virtualisation and pre-virtualisation approaches have the potential to provide better performance than native virtualisation. For this reason the vNUMA implementation does not utilise VT-i. Even with VT-i, some architectural features, particularly those described in Sections 3.4 and 3.5, remain difficult to virtualise efficiently and demand elaborate workarounds.

| Instruction | Function |
|---|---|
| `cover` | Allocate new register window, possibly save size of old |
| `mov =ar.rsc` | Read RSE control register |
| `thash/ttag` | Calculate page table hash and tag values |
| `fc` | Flush cache line |
| `mov =cpuid` | Read CPU type and feature information |
| `mov =pmd` | Read performance monitoring counter |
| `mov =ar.k*` | Read kernel register |

**Table 3.1:** Sensitive Itanium instructions

## 3.1 Sensitive instructions

As mentioned previously, virtualisation involves running an operating system in a less privileged mode. Privileged instructions trap when they are executed in the less privileged mode, and can then be emulated by the virtual machine monitor. Unprivileged instructions generally execute normally. However, many architectures have unprivileged instructions which do not trap but somehow behave differently in the less privileged mode; these are known as *sensitive instructions* [79]. These are problematic because it is difficult for the virtual machine monitor to intercept them in order to correct their behaviour. The problematic instructions in the Itanium architecture are listed in Table 3.1 and described in detail in this section.

For example, Itanium provides an instruction called `cover` which is used in conjunction with the register windowing feature. The specific function of the `cover` instruction is to hide the current register window — *covering* it — and to allocate a new empty window.

Since control of register windows is in the hands of applications, the `cover` instruction is unprivileged. Unfortunately, the designers saw it fit to overload the instruction with another side-effect: in the special case that interruption collection is disabled[1], it also saves the original size of the register window to a control register. This functionality is used by operating systems in order to save register

[1]Interruption collection is not the same as interrupt control, but the two control bits are closely related. Interruption collection determines whether context is saved to control registers when a new exception occurs, or whether the processor attempts to keep control registers intact. With interruption collection off, it is not possible to return to the original context after an exception.

window state during an exception, simultaneously hiding the old window and obtaining its size. In a virtualised environment, it is not safe to allow an operating system to disable interruption collection, so `cover` always executes using its unprivileged semantics. Namely, it hides the old frame but does not write its size to the control register (and does not trap to allow the hypervisor to save the size). Thus, even though the hypervisor can intercept the later read of the control register, the size of the old frame has been irrecoverably lost.

The register stack engine control register, `ar.rsc`, is also exposed to userspace. This is problematic for virtualisation because it contains a field that specifies the privilege level with which RSE accesses occur. Normally this cannot be used for privilege escalation, since writing a value that is more privileged than the current privilege level is never allowed; such values are silently clipped to the current privilege level. However, in a virtualised environment, reads expose the actual privilege level rather than the virtualised privilege level, which may or may not affect correct execution. Of most concern, naïve writes by privileged code may establish incorrect values for less privileged code. For example, consider if an operating system kernel attempts to confine code to ring 1, and therefore specifies a value of 1 in the privilege level field. If the ring compression technique (Section 3.2) is used, this will inadvertantly give the code access to the ring where the kernel itself is. Fortunately, existing Itanium operating systems only utilise rings 0 and 3, so this never occurs.

The `thash` and `ttag` instructions are used to calculate the address and tag value, respectively, for an entry in the Itanium architectural page table (which is a type of hashed page table [42]). While it would seem that these instructions have little use to application code, they only perform calculations and thus processor designers deemed that they could safely be made unprivileged. However, in order to perform their calculations, these instructions actually utilise privileged state information, namely the page table base address, size and format. When a guest operating system executes these instructions, it expects the calculations to be made in the context of its own page table. However, in actual fact, the instructions will silently return information about the real page table used by hardware, namely the virtual machine monitor's page table!

The `fc` instruction to flush cache lines also has a curious quirk, albeit

somewhat less problematic. When executed at the most privileged level (ring 0), it bypasses TLB protection checks. At lower privilege levels, such as when the kernel is demoted to ring 1 in a virtualised environment, `fc` performs permission checking as if it was a one-byte read. Attempts to flush execute-only pages now result in memory protection faults. In this case the virtual machine monitor must be very careful to emulate the `fc` instruction rather than delivering the protection fault to the guest operating system, which may be fatal. To avoid extra checks in the page fault path, it may be desirable to deal with `fc` by replacing it in the same way as other sensitive instructions.

The `cpuid` register file is an array of registers providing identification information about the processor, such as the vendor, model, architecture revision and supported features. Access to this register file does not require any special privileges; thus the real processor's identification information is read directly by the guest operating system and applications. In most cases this is not a problem, since the processor features described by the Itanium's `cpuid` registers primarily describe the availability of user-level instructions. However, a guest operating system may utilise the model information to tune its functionality in undesirable ways; this may cause problems when a virtual machine is migrated to a different type of processor. Additionally, there are situations where the virtual machine monitor may want to avoid advertising certain features (for instance, vNUMA does not currently support 16-byte atomic memory operations), and it would be desirable to be able to reflect this in `cpuid` values.

The `pmd` register file provides access to performance monitoring counters. Within a virtual machine, it might be desirable to virtualise these counters, so as to isolate the operating system from the processor that it is executing on. Unfortunately, even if the counters are configured to deny userspace accesses, unauthorised reads do not fault and simply return zero values. This makes it difficult for a hypervisor to intercept and emulate these accesses. If performance monitoring functionality is to be exported to the guest operating system, the counters must directly correspond to those provided by the underlying processor, and their configuration and values must be context-switched as part of the virtual machine state.

Finally, there is a set of *kernel registers* which can only be written by

privileged code but are readable at any privilege level. This provides a lightweight mechanism to convey certain data from a kernel to applications, although in practice these registers are mainly used by operating systems to store processor-local kernel data. Since reads of these registers never fault, at any privilege level, they cannot be virtualised in the normal way. In Type I virtual machine monitors, the registers can be context-switched between virtual machines, but this proves problematic for Type II VMMs since a host kernel will use the kernel registers for its own internal purposes. The guest kernel then unavoidably observes the values written by the host kernel, instead of the virtual values.

There are a number of approaches for dealing with sensitive instructions. Generally these can be categorised according to whether they detect the sensitive instructions statically within a kernel binary or dynamically at runtime, and whether they attempt to translate them intelligently or simply allow them to fault to the hypervisor for handling.

Early versions of vBlades, Xen/ia64 and vNUMA have all used simple tools which statically pre-process a kernel binary, replacing sensitive instructions such that they fault. The replacement instructions are chosen from the set of privileged or reserved instructions that are not normally used by an operating system, and that fit in the same space as the original instruction. Such substitutions can also be performed dynamically at runtime[2], or on newer processors the instructions can be intercepted at runtime using the VT-i extensions: VT-i provides a special processor mode in which any attempt to execute a sensitive instructions results in a fault. Such techniques are simple and reliable, but do not provide optimal performance, since each sensitive instruction now causes a fault and is emulated by the hypervisor.

Later versions of Xen/ia64 use manual para-virtualisation, relying on manual modifications to the operating system that replace the sensitive instructions. To avoid the associated engineering effort, the current version of vNUMA instead uses the automated pre-virtualisation technique described in Chapter 4, which substitutes the sensitive instructions as they pass through the assembler. In both

[2] A good technique, described by Lawton [57], is to translate a page at the time it is inserted into the instruction TLB, while inserting the original version of the page into the data TLB. This ensures that the operating system can never observe the modified pages, even if the translator inadvertantly misidentifies data as instructions.

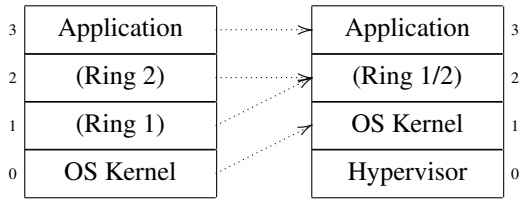



**Figure 3.1:** Ring compression

cases, the changes are made statically; since symbolic information is still present at build time, replacing a single instruction with larger emulation code rarely poses problems. Performing this feat at runtime, on the other hand, would be far more difficult.

## 3.2 Ring compression

The Itanium architecture, like x86, provides four hierarchical privilege levels or *rings*, derived from the model introduced in Multics [18]. These privilege levels are numbered from 0 to 3, with 0 being the most privileged and 3 the least.

Since code executing in ring 0 is generally allowed to execute any privileged instruction, an operating system kernel must be demoted from ring 0 to ring 1 to enforce virtualisation. However, this now only leaves three rings for the operating system to use, instead of four. If the original operating system were to rely on all four of those rings, it would be necessary for the hypervisor to merge two of them together, as shown in Figure 3.1. This results in a loss of protection between those two rings. Fortunately, in practice the majority of operating systems do not use ring 1 and 2 at all, so this does not present a problem.

In processors with Intel VT-i, ring compression is unnecessary. This is because, with VT-i, privileged instructions always fault when the processor is executing in virtual machine mode, *even in ring 0*. Thus it is safe to execute the guest operating system in ring 0, providing it with the full set of rings.

## 3.3 Hiding the VMM

When the Itanium processor encounters an exception, it simply branches to the appropriate exception vector address, without switching off virtual address translation. This means that a hypervisor's exception vectors must be located somewhere in the virtual address space. However, it is not obvious where to put them; the guest operating system believes that it has control of the entire virtual address space, and may attempt to insert mappings in any portion of it.

One approach is to optimistically choose a location where guest operating systems are unlikely to place mappings. If a guest operating system does attempt to place a mapping there, the exception vectors can be remapped to a new location.

An even simpler approach is to leverage the fact that the Itanium architecture defines a variable number of implemented virtual address bits, from a minimum of 51 bits per region to a maximum of 61 bits per region (which, over 8 regions, covers the entire 64 bit address space). At boot time, an operating system must query the number of implemented address bits. When executing on a processor with more than 51 implemented bits (any Itanium processor other than Itanium 1), the hypervisor can report that one less address bit is implemented. The guest operating system must not attempt to place mappings in the part of the address space that is reported as unimplemented, and this can be enforced by the hypervisor. The hypervisor can then be hidden within that space.

In fact, VT-i enforces this one-bit differential. When executing in virtual machine mode, the processor behaves as if one less bit was implemented, and guarantees that the guest operating system cannot access the hidden region. When an exception occurs, the virtual machine mode is exited, revealing this hidden address space where the hypervisor is concealed. This is a necessary part of VT-i because memory protections would not otherwise protect the hypervisor from the operating system, since they both execute in ring 0.

## 3.4 Register stack engine faults

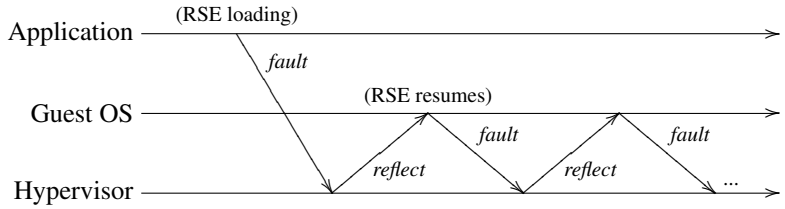
The register stack engine typically operates through virtual memory, and therefore may encounter virtual memory faults while loading or storing registers. When

this happens, RSE operation may be halted partway through loading a frame. The exception handler then executes with an *incomplete* register stack frame, some of which may be undefined; this is only possible in ring 0. When the handler has finished, it executes the *return from interruption* (`rfi`) instruction, which returns to the original context and causes the RSE to resume loading the current frame.

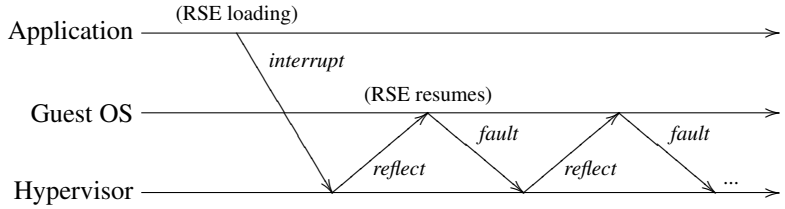
In a virtualised environment, the hypervisor will likely desire to reflect the virtual memory fault to the guest operating system for handling. However, in order to invoke the guest operating system in ring 1, the hypervisor must execute an `rfi` instruction. Even though it is now returning to a different privilege level and IP address, the `rfi` instruction nonetheless causes the RSE to resume loading the current frame and the fault is raised again (Figure 3.2(a)).

The workaround used in vNUMA and all known Itanium hypervisors is to emulate a `cover` instruction in the hypervisor when an incomplete frame is detected. This `cover` hides the current incomplete frame and instead allocates a new zero-size frame for the operating system exception handler. To the operating system, it is as if the faulting userspace application was executing with a zero-size frame. Typically the operating system handles the fault and executes `rfi` to return to the application, at which point the hypervisor restores the original frame (Figure 3.2(c)). Alternatively, the operating system may desire to save the userspace state, in order to enter C code or switch processes. In that case it necessarily executes `cover`, which is required to obtain the size of the userspace frame. Now this latter `cover` can be ignored since it has already been performed (Figure 3.2(d)).

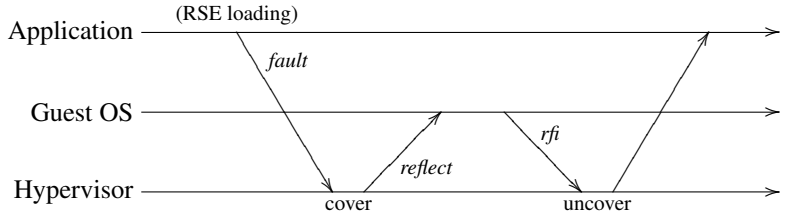
It should be noted that, in addition to the case where loading of an RSE frame is interrupted by a memory management fault, the frame load can also be interrupted by the arrival of a hardware interrupt (Figure 3.2(b)). After the hypervisor reflects the interrupt to the guest operating system, the RSE resumes attempting to load the frame. If the RSE faults at this point, the fault appears to come from the interrupt vector address; such nested faults are usually fatal. The `cover` solution must also be applied to this scenario.



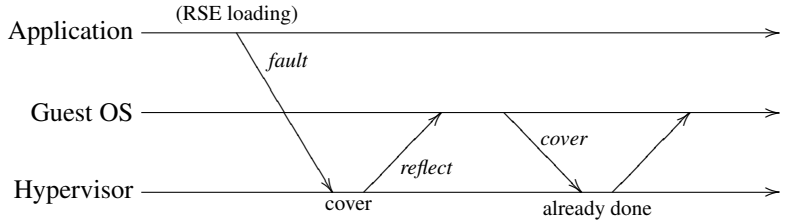

(a) Reflecting an RSE fault results in nested RSE faults




(b) Hardware interrupts can also trigger nested RSE faults




(c) `cover` workaround; guest OS handles exception and returns (`rfi`)




(d) `cover` workaround; guest OS then executes `cover` in order to save state

**Figure 3.2:** Register stack engine virtualisation: problem and workaround

## 3.5 Complex translation modes

When building a virtual machine monitor, it becomes necessary to virtualise the physical memory of the virtual machine. A virtual machine must never actually run with address translation disabled, or it could subvert all memory protection. Instead, when the operating system disables address translation, a VMM tears down the existing virtual address space and establishes a set of virtual mappings that provide the illusion of a physical memory space.

An obscure feature of the Itanium architecture is the ability to separately enable and disable virtual translation for instruction, data and register-stack-engine accesses. For example, it is possible to arrange for instruction and data accesses to be physically addressed (bypassing the TLB), while RSE accesses are translated virtually, or any other combination. Thus, a data access to address $X$ may access physical address $X$, while a RSE access to address $X$ may access some other page. This poses complications for virtualisation.

It is theoretically possible to provide separate mappings for instruction and data references, since the architecture provides for separate instruction and data TLBs. This requires disabling hardware-assisted TLB reload, since it is not possible to provide this distinction in either of the architectural page table formats. Another possibility is to restrict permissions on each mapping to only allow either data or instruction accesses, but not both, and emulate any instructions that require incompatible mappings. However, all of these options are complex and do not provide a mechanism to differentiate between data and RSE references, short of emulating all data references. Unfortunately VT-i is of no help in the matter.

This is not purely an academic question; Itanium Linux regularly disables data translation in order to access page tables, while continuing to execute instructions virtually. It also happens to leave register stack translation enabled, although in that particular case it has no effect on the outcome.

The solution used in current hypervisors is to expose the virtualised physical address space in the bottom region of the address space if *any* of the physical translation modes are enabled, while leaving the virtual address space active in higher regions to satisfy other types of accesses. This works for Linux, but is not a general solution. If an operating system kernel was to place virtual mappings in

the bottom region of the address space, and then use complex translation modes, this approach might be rendered invalid.

# Chapter 4

# Pre-virtualisation

As previously mentioned, using the trap-and-emulate method for all privileged and sensitive instructions presents considerable performance overhead. This can be addressed by para-virtualisation — modifying the guest operating system — but to do this manually requires significant engineering effort and is error-prone. Pre-virtualisation is a method by which much of this para-virtualisation effort, namely substitution of sensitive instructions with code tailored for a virtual machine, can be performed automatically. This section will first describe the pre-virtualisation technique (which was developed in collaboration with others [60]), and then specifically address the challenges in applying it to the Itanium architecture.

## 4.1 Mechanism

The source code of an operating system kernel is typically a combination of assembly language and a higher-level language such as C. At compile time, the C code is also translated into assembly language. Thus, at some stage all operating system code passes through the assembly language stage. This is the ideal level to perform instruction transformations, because much symbolic information is still present at this level: control flow transfers are performed via named labels rather than fixed addresses. This means that it is generally possible to substitute individual instructions with longer emulation sequences, without affecting control flow.

The pre-virtualisation process involves replacing the assembler with a wrapper that performs certain replacements before invoking the real assembler. The first stage is to translate the privileged and sensitive instructions into macro calls. For example, a privileged instruction `(p6) mov r16=cr18` — which reads control register 18 into general register `r16`, conditional on predicate register `p6` — might be replaced by the macro call:

```
emul_read_cr pr=p6,dest=r16,cr=18                    (1)
```

This makes further substitution simpler, since all instructions are now in a common format that can be expanded by the assembler's macro facilities.

Next, the assembler is invoked, together with a macro definition file that provides an implementation of each macro. In the simplest case, these macros target a specific hypervisor. For example the following provides an implementation of `emul_read_cr` for vNUMA, which reads the value of the control register from a shared memory location:

```
.macro emul_read_cr pr,dest,cr
(\pr)   mov \dest=__vnuma_cpu+CR0_OFFSET+(8*\cr)
        ;;                                           (2)
(\pr)   ld8 \dest=[\dest]
.endm
```

Such substitutions result in more efficient virtualisation than a trap-and-emulate approach. Many instructions can be directly replaced with simple emulation code, avoiding the overhead of a trap and the complexity of reading, decoding and simulating the instruction. Those which cannot be emulated completely in user mode can access hypervisor services via the most efficient interface possible. In particular, the Itanium architecture provides an `epc` instruction (*enter privileged code*) which allows zero-overhead entry to privileged mode; vNUMA exports a number of services in this way, which can be leveraged by the macro definitions.

However, a binary so compiled will only work on the specific hypervisor targeted. A more sophisticated possibility (not yet implemented for vNUMA) is

to insert the original instruction back into the binary, but add padding afterwards, as follows:

```
.macro emul_read_cr pr,dest,cr
(\pr)   mov \dest=cr\cr
(\pr)   nop.m 0x0
.endm
```
(3)

The resulting binary contains the original instruction and will therefore execute correctly on real hardware. The padding no-ops can generally be executed in parallel with the instruction, so the only impact is a small increase in code size.

When loaded on a hypervisor, however, the hypervisor can easily substitute the replacement instructions, since there is sufficient space to do so. For example, on vNUMA the control register read and the padding might be replaced with the `mov` and `ld8` sequence in example (2) above. In order to aid the hypervisor in locating and patching these instructions, it is helpful to record the beginning and end of each patch target in a special section, which can be achieved by a macro definition such as the following:

```
.macro emul_read_cr pr,dest,cr
9998:
(\pr)   mov \dest=cr\cr
(\pr)   nop.m 0x0
9999:
.pushsection .afterburn
.quad 9998b
.quad 9999b
.popsection
.endm
```
(4)

It is important to ensure that all code that originally executes in kernel mode is pre-virtualised so that the sensitive instructions are replaced. In particular, any dynamically loaded modules must also be built using the pre-virtualising assembler.

## 4.2 Challenges

While this idea of substituting instructions with emulation code is conceptually simple, the specifics of various processor architectures present different challenges. Previous work has only addressed the x86 architecture, which has a number of features that make pre-virtualisation particularly simple; for example, it is always possible to save registers on the stack, even in low-level operating system code. In this section the challenges associated with Itanium pre-virtualisation will be described, and some solutions presented. Many of the same issues also apply to other non-x86 architectures.

### 4.2.1 Scratch registers

The emulation code frequently requires spare registers in order to perform intermediate calculations, call an external procedure, or simply access memory[1]. However, it is non-trivial and sometimes impossible to determine which registers are safe to overwrite in a given context; register usage conventions may not apply in handwritten assembly code. It is also not possible to save and restore registers to memory without having at least one scratch register to start with (in order to generate the memory address); it is not safe to rely on the stack pointer since it may be invalid in low-level code.

When substituting instructions that write a destination register, the destination register can be considered scratch until the final result is generated. This is the case in the `emul_read_cr` case in example (2) above, where the destination register has been used to temporarily store the load address. However, many instructions do not have a destination-register operand. It would also be preferable to have more than one scratch register available, to avoid costly saving and restoring of further needed registers.

The solution used in vNUMA is to virtualise a subset of the machine registers that are rarely used, specifically `r4-r7` and `b2`, when executing guest kernel code (which should exactly correspond to the body of code that is pre-virtualised). All references to those registers in the guest kernel code are replaced with instruction

[1] Itanium memory access instructions require the target address to be in a register.

sequences that read and write shadow copies of those registers in memory, making the real registers available for the emulation code itself. When transitioning from guest user mode to guest kernel mode, the hypervisor saves the real registers to the memory-backed shadow registers; this allows the guest kernel to access the user mode values of the registers. Then, when returning to guest user mode, the shadow register values are restored into the real registers[2]. This provides complete transparency; the guest operating system is never aware of the conceit.

Since one of the most common functions of the emulation code is to access and update the virtual CPU state, it is useful to have a pointer to that state available at all times. One of the scratch registers is reserved for that purpose, and is initialised by the hypervisor upon entry to the pre-virtualised guest kernel.

### 4.2.2 Atomicity

Instruction rewriting replaces a single, non-interruptible instruction with an instruction sequence. If the emulation sequence updates state non-atomically and an interrupt arrives during that sequence, the interrupt handler could execute with the virtual CPU in some undefined state. Worse, the interrupt would clobber the values of scratch registers, since the OS interrupt handler itself contains pre-virtualised code. This would make the scratch registers useless in practice. There are two possible solutions: deferring interrupts during a block of emulation code, or attempting recovery if an interrupt arrives. Both options involve complications.

Initially the latter option was trialled; an emulation block would set one of the scratch registers to the address of a roll-back point, and clear that register at the end of the block. If an interrupt was delivered, the hypervisor would override the instruction pointer reported to the interrupt handler, such that upon return execution would be restarted from the roll-back point. This works well for emulation sequences that simply perform calculations before committing state, because it is safe to perform the calculations again. However, even in this case

[2]The guest kernel may also be entered without hypervisor intervention, using the `epc` — *enter privileged code* — instruction; the emulation code for `epc` is then responsible for saving the registers. It is also necessary to restore the registers on exit; unfortunately exiting the kernel after an `epc` is done with a normal return instruction, which is difficult to detect and pre-virtualise automatically. The guest kernel was modified to annotate that particular return instruction.

there is the possibility that the state update might be applied twice, if an interrupt is delivered between the state update and the clearing of the roll-back register. Also, more complex emulation code can be difficult or impossible to write in a way that is restartable, making it necessary for these emulation sequences to explicitly disable and enable interrupts.

A better solution may be simply to provide a lightweight mechanism for deferring interrupts during complex emulation code. The `r4` register, which is normally used to store a pointer to the virtual CPU state, can simultaneously be used to store an interrupt deferral flag, avoiding the need for extra scratch registers. Consider if the hypervisor initialises `r4` to one byte before the virtual CPU structure. A critical section might then be constructed as follows:

```
add r4=1,r4
  ...                                                    (5)
add r4=-1,r4
```

Within the critical section, `r4` points to the virtual CPU structure; outside the critical section, `r4` points one byte below it. In either case the members can be accessed using offsets, and the hypervisor can easily determine whether interrupts should be deferred by checking the lowest bit of `r4`.

However, this is not sufficient; there needs to be a lightweight mechanism by which interrupts are finally delivered when the critical section completes. A number of options are possible:

- The hypervisor could re-attempt delivery of the interrupt after a certain time period. However, this has the usual drawbacks of a timed backoff approach: if the timer is set too short, the critical section may not complete before timer fires; if the timer is set too long, interrupt latency suffers.

- The hypervisor could write to a predicate register to indicate a pending interrupt. This predicate register can then be tested in a single instruction at the close of the critical section:

```
add r4=1,r4
  ...
(p63) break 0x1234  // invoke hypervisor if p63 true          (6)
add r4=-1,r4
```

While simple, this solution is less attractive since it necessitates hijacking a predicate register from the operating system, and requires an additional instruction at the end of each critical section.

- The hypervisor could set the NaT bit[3] on the `r4` register to indicate a pending interrupt. Certain instructions that 'consume' that register will then cause a fault if and only if the NaT bit is set:

```
add r4=1,r4
  ...                                                    (7)
lfetch.fault [r4],-1
```

Here `lfetch` (cache line prefetch) is chosen because it has a post-decrement form that obviates the need for an extra instruction. The problem with this approach is that it is no longer possible to use `r4` within the critical section, because it may acquire a NaT bit at any time (making it unsafe to use); thus it would become necessary to use an additional register.

- The hypervisor could change virtual memory permissions to indicate a pending interrupt, in order to force a final store to fault:

```
add r4=1,r4
  ...                                                    (8)
st1 [r4]=r0,-1
```

Once again the post-decrement form of store is used to avoid an extra instruction. Of course, if other instructions in the critical section were to write to the same page, an undesired fault might occur. To avoid this, the virtual CPU structure could be spanned over two pages, with permissions only removed from one; the other possibility is to use the processor's debug functionality to only trap on writes to a certain word while leaving the rest of the page writable.

[3] NaT bits are an additional bit maintained with each register, used for data speculation.

- The hypervisor can use the Itanium's performance monitoring unit to detect the final instruction of the critical section. This method requires no changes to the original critical section code:

```
add r4=1,r4
  ...
add r4=-1,r4
```
(9)

  The Itanium performance monitoring unit includes an *opcode matcher* which allows specific instructions to be counted and, optionally, trigger an interrupt. In this case, when an interrupt needs to be deferred, the opcode matcher can be programmed to cause an interrupt when the `add r4=-1,r4` is executed[4], which is precisely what is desired. This method is simple and works well.

### 4.2.3 IP-inspecting code

While rare, kernel code may occasionally inspect the instruction pointer and make assumptions about code layout which are violated by pre-virtualisation. There is one such instance in Linux, which is similar to the following:

```
L1:
  ssm psr.i|psr.ic
  mov r16=ip
  ;;
  add r16=L2-L1,r16
L2:
```
(10)

The first instruction (`ssm`) enables interrupts. Then, the absolute address of label `L2` is calculated by reading the current instruction pointer (`mov r16=ip`) and adding the offset `L2-L1`. Itanium instructions are contained in bundles of three with a single instruction pointer value, and this code implicitly makes the assumption that the instruction pointer read (`mov r16=ip`) is contained in the

[4]Technically, the filter that is programmed matches any instruction of the form `add r4=`*`immediate`*`,`*`register`*, since that is the most specific form possible. However, this is without consequence; there should be no other matching instructions within the critical section.

same bundle as the first instruction, and therefore returns the address of `L1`. However, when the `ssm` instruction is substituted with larger emulation code, this is no longer the case; the instruction pointer read is now contained in a subsequent bundle and returns the address of that bundle.

This code can easily be modified so that it still functions identically on real hardware and yet avoids making the layout assumption that is problematic for pre-virtualisation. The obvious solution might be to place the label against the instruction pointer read, or to place the instruction pointer read before the `ssm` instruction; however, these both result in a slight code expansion because of the specific way in which Itanium instructions are encoded in bundles. It is possible to avoid this by using a *tag* within brackets, as follows:

```
  ssm psr.i|psr.ic
[L1:]  mov r16=ip
   ;;                                                    (11)
   add r16=L2+2-L1,r16
L2:
```

The tag functions like a label but does not force the assembler to start a new bundle; however it also has the repercussion that L1 is now an intra-bundle address, so some corrective arithmetic is necessary. The above code is correct as long as the lowest 4 bits of the result is eventually ignored (which is generally true since the lowest 4 bits of an IP address are ignored by the processor). A more elegant way would be to round L1 down to a bundle boundary by masking out the lowest 4 bits; unfortunately only limited arithmetic on labels is possible in assembly code.

### 4.2.4 Code expansion

Substituting one instruction with multiple instructions causes an increase in code size. This may be problematic if certain basic blocks are limited to an absolute size. For example, this is an issue for the Itanium exception vector table, which contains fixed size entries of either 256 or 1024 bytes (depending on the vector). In practice, by carefully implementing the macros used in such contexts to minimize code expansion, it was possible to avoid overflowing the Linux vectors. In rare

cases where this is too difficult, it may be necessary to modify the source code to relocate large vectors out-of-line, replacing them with a branch instruction. Since this is a simple transformation, it could also be done automatically by a tool, given the right annotations.

# Chapter 5

# Hypervisor architecture

The initial prototype of vNUMA was constructed on top of Linux as a Type II VMM. This Linux substrate provided a rich development and debugging environment, in which the challenges involved in virtualising the Itanium architecture could be investigated. To achieve maximum possible performance, vNUMA was later adapted into a Type I VMM, executing in privileged mode without an underlying operating system. However, the userspace version was not abandoned and evolved into a separate project by the name of Linux-on-Linux [13], which continues to be improved. The basic design of the userspace implementation is described in Section 5.1, while Section 5.2 describes the corresponding design of the standalone implementation.

## 5.1 Userspace (Linux-on-Linux)

In principle, it is quite straightforward to build a virtual machine monitor atop Linux: the guest kernel code is simply loaded into a Linux process and executed from the start address. The majority of instructions — such as arithmetic, memory, or branch instructions — function identically in this userspace environment. When a privileged instruction or other exception is encountered, a signal is delivered to the process; the core of the virtual machine monitor is then embodied in a set of signal handlers.

The signals hooked by the Linux-on-Linux VMM are listed in Table 5.1. Whenever one of these signal handlers is invoked, Linux saves the register state

| Signal | Function |
|---|---|
| SIGILL | privileged instructions |
| SIGSEGV | virtual memory faults |
| SIGFPE | floating point exceptions |
| SIGTRAP | simulator system calls (device interface) |
| SIGUSR1 | redirected Linux system calls (see Section 5.1.2) |
| SIGVTALRM | timer interrupts |
| SIGIO | asynchronous IO (console, network) |

**Table 5.1:** UNIX signals used by userspace VMM

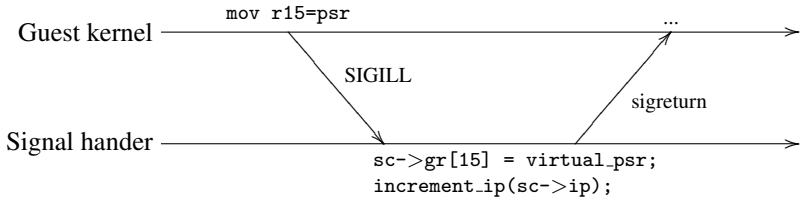



**Figure 5.1:** Emulation of a privileged instruction (move from processor status register to `r15`). `sc` is a parameter to the signal handler that points to saved register state.

on the userspace stack[1] and passes a pointer to this save area as an argument to the signal handler. The signal handler can then use and update the register state in order to emulate the desired behaviour, similarly to a standalone virtual machine monitor. For example, handling of a privileged register read is shown in Figure 5.1: the faulting instruction is inspected, the virtualised value of the privileged register is written to the target register (via the save area), and the instruction pointer is incremented in order to skip to the next instruction. Finally, returning from the signal handler invokes the `sigreturn` system call which restores the register state from the save area and resumes execution.

A device and firmware environment must also be provided for the guest operating system. Fortuitously, Itanium Linux includes a set of simple device drivers and firmware stubs which use a system-call-like interface; these are

[1]The standard `sigaltstack` system call is used to provide a separate stack for the signal handlers, as the guest stack pointer may not always be valid.

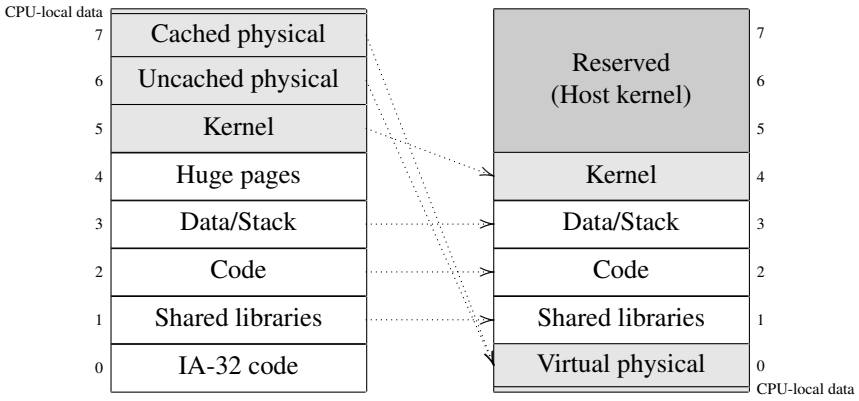



**Figure 5.2:** Relocating a guest kernel to avoid conflicting with a host kernel. The shaded regions are kernel areas; unshaded regions are available to applications.

normally used in conjunction with HP's `ski` simulator [16]. This interface can be leveraged to greatly simplify the VMM implementation, as compared to the complexity of emulating real memory-mapped devices and firmware interfaces. In Linux-on-Linux, the underlying devices are naturally implemented in terms of UNIX devices — the console device is mapped to standard input and output, the disk driver simply reads and writes from a flat file on disk, and a virtual Ethernet device is created using Linux's TUN/TAP driver [54].

### 5.1.1 Address space conflicts

In practice, there are a number of other issues that must be addressed. One of the most problematic is the fact that the address space layout of the guest kernel is likely to conflict with that of the host kernel. For example, the guest kernel will desire to map its code or data at some fixed virtual address and refer to it through that address. However, if the host kernel also uses that same area for its code or data, then the userspace virtual machine monitor will be forbidden from placing mappings there. Ultimately it is not possible to resolve this transparently (short of individually emulating all guest kernel instructions, which would be very slow); it is necessary to modify the guest operating system to avoid the conflict.

The Itanium architecture provides a 64-bit virtual address space divided into

eight regions; each of these regions is a window into a larger conceptual address space, providing a limited form of segmentation. Linux's address space usage is based around the eight regions, as shown in Figure 5.2. The bottom five regions are available to applications, including the userspace virtual machine monitor (which to the operating system is simply another application). Region 0 is primarily used for IA-32 applications, although it is also available to native Itanium applications if they were to specifically request a mapping there. Region 1 is the default space for mappings returned by the `mmap` system call; shared libraries and other mapped files are located there. Application code, data and stack pages are placed in regions 2 and 3. Region 4 is assigned to a rarely used feature known as *huge pages*, whereby applications can request large regions of contiguous memory that are mapped using a large page size (to minimise TLB footprint). The top three regions are reserved by the host kernel. Kernel code and data are mapped into region 5. Regions 6 and 7 represent identity mappings of the physical address space (the one difference being that the mappings in region 6 are set up to bypass processor caches, for the benefit of device drivers). The very top of region 7 is used for CPU-local kernel data; this location is chosen because the address is equivalent to a small negative offset from zero.

In order to allow the guest kernel to execute within a userspace process, it is necessary to modify it so that it only uses the lowest five regions, while maintaining compatibility with the majority of applications. To achieve this, the guest kernel is moved down from region 5 to region 4, as shown in the figure. Of course, this is not compatible with the huge page feature, in either host or guest kernel; however that feature is rarely used. The CPU-local data is moved to the bottom of region 0, so now it is accessed using a positive offset from zero rather than a negative offset.

The one issue that remains is to provide a mechanism for the guest kernel to access (virtualised) physical addresses. Normally there are two ways that the kernel accesses physical addresses: via the identity-mapped regions (6 and 7), or by disabling address translation and accessing physical addresses starting at 0. If that same bottom region of the address space is also used for the identity-mapped region, then switches between the two modes can essentially be disregarded by the virtual machine monitor, greatly decreasing their cost. The compromise made is

that applications that explicitly require region 0 (IA-32 applications and some Java virtual machines) will not function correctly. If this compromise was found to be unacceptable, region 1 could instead be hijacked for virtual identity mappings by merging the `mmap` region into region 2 or 3. In either case there is only one region needed, not two as on real hardware; even if one was to directly map real devices into the virtual machine, the virtual machine monitor can and should arrange for the device memory to be mapped with the correct attributes, regardless of the attributes specified by the guest kernel.

### 5.1.2 System call redirection

A namespace collision also occurs for system calls. Assuming that the host and guest operating systems use the same instruction to invoke a system call, any system call will be handled directly by the host kernel. However, within the virtual machine, the system calls must be redirected to the guest kernel. This can be achieved without modifying the host kernel by using the `ptrace` mechanism, which notifies another process whenever a system call is entered or exited. While the tracing process cannot abort the system call, it can adjust the register state of the virtual machine process so that it instead executes a harmless system call such as `getpid`. It simultaneously sends a signal (such as the user-defined signal `SIGUSR1`) to the virtual machine process, which then invokes the guest kernel to handle the system call. This process is shown in Figure 5.3.

However, as implied by the diagram, this `ptrace` redirection mechanism is rather slow, requiring a context switch to the tracing process and a number of extra system calls. This is exacerbated by the fact that, as a side-effect of `ptrace` semantics, the tracing process also synchronously receives notification of all signals delivered to the virtual machine, as well as system calls executed within the VMM signal handlers — none of which it needs. More recent versions of Linux-on-Linux modify the host kernel in order to provide more efficient primitives for system call redirection. For example, the virtual machine monitor can arrange for a particular user-specified signal to be delivered whenever a system call is executed with that signal unmasked; within the VMM, the signal is masked, allowing system calls made by the VMM to execute. This avoids the costly side-

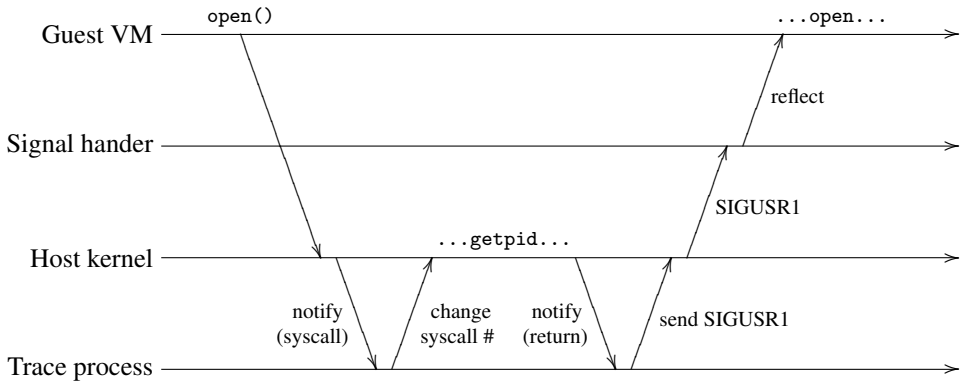



**Figure 5.3:** System call reflection using `ptrace` facility

effects of `ptrace`.

### 5.1.3 Other issues

Itanium Linux — like many operating systems — manages floating point state in a lazy manner. Upon a context switch, floating point registers are disabled. If the new process does not use those floating point registers, the values are intact for when the first process resumes execution, without the cost of saving and restoring. If the new process does happens to use those floating point registers, a fault occurs; the operating system can then save the old registers and establish values for the new process[2]. Unfortunately, within the userspace virtual machine monitor, there is no way to disable floating point registers and receive a signal notification when they are accessed; Linux provides applications with the illusion that all registers are permanently enabled. It is therefore necessary to either force the saves and restores to be performed eagerly — by immediately delivering a fault after switching to a process with floating point disabled — or to modify the host kernel to provide the required functionality. The latter was implemented by

[2]To be precise, a subset of the floating point registers (f2–f31) are saved and restored eagerly, while the remaining registers (f32–f127) are switched lazily in the manner described.

adding a mechanism that disables floating point registers and arranges for delivery of a `SIGFPE` signal if those registers are accessed.

Virtual memory emulation also presents both overheads and limitations. In a userspace virtual machine monitor, virtual memory mappings can be established by using the `mmap` system call to map parts of a file, representing physical memory, into the address space. However, a major limitation of the UNIX address space model is that there is no support for multiple address spaces or protection levels within a process. This means that, when the guest kernel switches address spaces, the entire address space must be torn down with `munmap` and then re-established (whether eagerly or lazily). Additionally, there is no memory protection between the VMM, guest kernel and guest applications. This could theoretically be solved by using multiple processes, but this would introduce significant overheads co-ordinating the processes and transferring state between them.

## 5.2 Standalone

Considering the various limitations of a userspace implementation without intrusive changes to the host Linux, vNUMA was eventually ported to run standalone and not depend on a host operating system. The standalone version of vNUMA implements the Linux boot protocol and can therefore be started from the Linux loader (`elilo`) in the same way that a Linux kernel would normally be booted. The loader is also responsible for preloading the guest kernel binary, via its `initrd` (initial RAM disk) facility. Once vNUMA starts, it installs its own exception vector table, sets up memory management structures, and finally switches to user mode to start executing the guest kernel. Instead of receiving exception notification via UNIX signals, vNUMA now has full control of the processor and handles exceptions directly.

### 5.2.1 Lightweight C environment

Absolute optimal performance might be achieved by writing the entire hypervisor in hand-tuned assembly code, however the complexity of vNUMA makes this impractical. Instead the vNUMA exception handlers are optimised with the

goal of entering and exiting a lightweight C environment as fast as possible, allowing the core of vNUMA to be written in C. The lightweight C environment differs slightly from the normal C environment. Most notably, in order to reduce the number of registers that need to be saved and restored, certain general registers and branch registers are not available; compiler-specific flags are used to prevent the compiler using those registers. While it might have been possible to specify those registers as callee-saved — saved and restored on demand — rather than forbidding their use completely, this would greatly complicate the process of determining their exception-time values (which is needed for instruction emulation). It is also of limited benefit, since it is in any case preferable that the compiler allocate additional registers on the register stack, leaving the job of saving and restoring registers to the processor. Floating point registers, and other special-purpose registers, are also not saved by the entry code, and must be explicitly saved and restored if they are to be used.

Additionally, the hypervisor code must be careful to avoid triggering processor exceptions, except in certain limited cases. This limitation further reduces the number of registers that must be saved. The Itanium register file includes a set of 16 *banked* registers, r16–r31, of which there exist two physical banks, 0 and 1. Normally code executes using bank 1, but when an exception occurs the bank 0 registers are made available for the exception handler, with bank 1 safely hidden. In vNUMA, these bank 0 registers are used for the hypervisor C code. If nested exceptions were allowed, the exception entry code would need to save bank 1 and switch to it, in order to make bank 0 available for additional exceptions. The exception code would also need to save and restore the various interruption state registers (instruction pointer, processor status, function state, etc.), since these would be overwritten by additional exceptions that occur; with one level of exceptions they can simply be left intact. On the other hand, disallowing nested exceptions places a greater burden on the programmer; any code that can cause an nested exception such as a TLB miss must conform to a special calling convention to be recoverable[3].

[3]Specifically, it is not possible for execution to continue after the point of the exception, since the processor does not record the instruction pointer (and other instantaneous state) to avoid overwriting the unsaved data about the outer exception. However, the return address register and the state of the parent function are available. Thus, the nested exception handler aborts the

This leaves only a small number of registers that need to be saved and restored by the exception entry/exit code. Its most important function, however, is to switch stacks: both the memory stack and the register stack. It is not safe for the hypervisor to execute on the userspace stack since the userspace stack pointer may be invalid or malicious. Switching the memory stack is trivial; it simply involves saving the old value of the stack pointer register and writing a new value. On the other hand, switching the register stack is a convoluted process that involves several instructions with non-trivial latencies. The scheduling in Figure 5.2 is believed to be optimal, and the resultant timing is around 36 cycles (not counting the fundamental overhead of vectoring to and from an exception, which is around 25 cycles); the remainder of the entry/exit code can be slotted in the latency gaps and add no extra overhead. Compared to the thousands of cycles for delivery of a Linux signal in the userspace VMM, this provides a much better foundation for building a high-performance system.

### 5.2.2 Memory management

Rather than being restricted to the UNIX memory model and `mmap`, an entirely new memory management layer was implemented for the standalone version of vNUMA. This new memory management layer directly leverages the hardware page table structures to provide the most optimal solutions with respect to the vNUMA system.

Like other virtual machine monitors, the vNUMA memory architecture can be thought of as having three layers: virtual memory that is used by applications within a virtual machine, the simulated physical memory of each virtual machine, and the real physical RAM of the host computer. The terminology for these addressing layers varies amongst the literature, for example:

- Virtual → physical → machine (VMware, Xen)
- Virtual → logical → physical (IBM POWER)

faulting function and arranges for the value 0 to be returned to the caller. To indicate adherence to this contract, a magic value must also be placed in a register when an exception is expected; this avoids strange bugs that might result if an inadvertant exception caused a function to return to its caller.

| Cycle | Code | Latency |
|---|---|---|
| 0 | Get `ar.rsc` (RSE control register) | 12 cycles to use |
| 1 | Set `ar.rsc = 0`[a] | 12 cycles to next RSE inst[b] |
| (3) | Get `ar.bspstore` (RSE store pointer) | 12 cycles to use |
| (5) | Set `ar.bspstore` to hypervisor stack | 5 cycles to next RSE inst |
| 13 | Get `ar.bsp` (new RSE frame pointer) | 12 cycles to use |
| 14 | Get `ar.rnat` (RSE NaT bits) | 5 cycles to use |
| | Call out to C code | |
| 0 | Calculate parameters for register reload using saved `ar.bsp` value[c] | 2 cycles |
| 2 | Set `ar.rsc` for register reload | 12 cycles to next RSE inst |
| 14 | Force reload of any spilled registers | 2 cycles min |
| 16 | Restore `ar.bspstore` | 5 cycles to next RSE inst |
| 21 | Restore `ar.rnat` | 1 cycle to next RSE inst |
| 22 | Restore `ar.rsc` | 12 cycles to next RSE inst[d] |
| | Return from interruption | |

[a] This places the RSE in enforced lazy mode, which is necessary to allow access to `ar.bspstore` and `ar.rnat`.

[b] When `ar.rsc` is written with an immediate constant, two additional instructions can be scheduled immediately afterwards with a lesser penalty. The details of this are complex and not relevant to this thesis.

[c] If any registers have been spilled onto the hypervisor register stack, they must be reloaded into registers before switching back to the operating system register stack.

[d] This should not have any impact as it is overlapped with the return.

**Table 5.2:** Switching register stacks

- Virtual → real → physical (SUN UltraSparc)
- Virtual → metaphysical → physical (vBlades, Xen/ia64)

The first option (virtual → physical → machine) makes sense from the point of view of an operating system, which continues to manage virtual to physical mappings and is oblivious to the machine addressing layer that has been added underneath it. However, it tends to create considerable confusion when describing the virtual machine monitor itself. Readers are familiar with the term 'physical memory' referring to the lowest layer: actual physical RAM. Confusing this terminology can lead to subtle bugs: for example, existing driver components leveraged by vNUMA refer to 'physical addresses', and it would be a serious error if a programmer accidentally provided a virtual machine address instead of

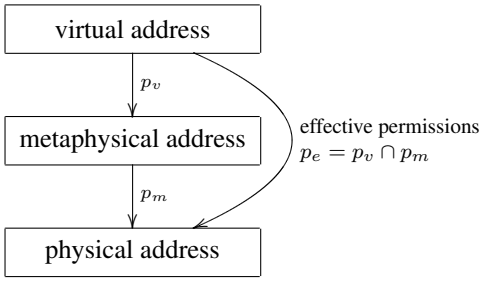



**Figure 5.4:** Address translation layers

a physical RAM address. Also, the term 'machine memory' is ambiguous; it is not immediately obvious whether it refers to the real machine or to a virtual machine (virtual → machine → physical might be equally sensible nomenclature, with machine memory being the memory of a virtual machine, so a reader must consciously remember which is intended).

Of the remaining options, both logical and real addressing have other specific meanings in the IA-32 architecture, and so are poor choices when attempting to avoid ambiguity. Hence, this thesis employs the last option: introducing the term *metaphysical memory* to describe the abstract memory space of the virtual machine, following the precedent set by publications related to vBlades and Xen for Itanium. *Physical memory* retains its normal meaning, referring to the physical RAM of the host system.

The virtual machine monitor is concerned with two levels of translation, as shown in Figure 5.4: from virtual addresses to metaphysical addresses (managed by the operating system and exposed through the operating system page tables), and from metaphysical addresses to physical addresses (known only to the VMM). The real TLB and the page table used by the processor (the *virtual hashed page table*, or VHPT) cache the composition of these two mappings — i.e. mapping virtual addresses directly to real physical addresses — so that applications can execute transparently.

The mapping from metaphysical to physical addresses is simply implemented via a flat array of mapping entries, since a virtual machine's metaphysical address space size is relatively small and bounded. The translation unit is fixed but

configurable; normally it is set to 4 KiB since that is desirable for the distributed shared memory system that will be described in Part II.

The operating system establishes virtual to metaphysical mappings via three mechanisms: explicitly inserting pinned mappings by using the `itr` instruction, explicitly inserting unpinned mappings by using the `itc` instruction, and implicitly inserting unpinned mappings by writing to its page table. Any pinned mappings *must* be maintained by the hypervisor in permanent data structures — they must be retained indefinitely and must never raise TLB misses to the operating system. However, the Itanium architecture only provides for a small number of these pinned mappings. The majority of virtual memory mappings are of the unpinned type; these are simply cached by a hypervisor in the same way they would be cached in a processor TLB. If these mappings expire from the cache and are required again, they can always be recovered from the operating system, either by raising a TLB miss or by consulting the operating system page table. Directly consulting the OS page table is clearly the more efficient method; to this end Itanium processors feature a *hardware page table walker* which allows the processor to directly access the OS page table (providing that it is in one of two supported formats). In vNUMA, the OS page table cannot be directly exposed to the processor, since vNUMA must enforce an additional layer of translation and protection. However, vNUMA internally implements a page table walker with similar semantics to the hardware walker; it can thus directly retrieve mappings from a guest OS that supports the hardware page table walker, without the expense of delivering a TLB miss.

vNUMA itself also exposes a page table in an appropriate format for the Itanium hardware page table walker. This page table is directly accessed by processor hardware as a logical extension of the TLB. In order to permit this, it must contain mappings from virtual addresses directly to physical addresses. (The downside of caching the composition of the mappings is that, like the TLB, it must be updated whenever either of the two component mappings change.)

The format that was chosen for this page table is the *long* page table format defined by the Itanium architecture, which is a type of *hashed page table* [42]. This page table format is fixed in size and supports multiple address spaces in the one global hash table, making it well suited to application as a virtual TLB in

vNUMA. In contrast, the Itanium *short* format requires separate page tables for each address space (nested within that address space); this would create greater complexity in memory management, and potentially greater inefficiency if the OS uses address spaces sparsely.

There are, however, also some disadvantages to this choice. As defined by the Itanium architecture, the long-format page table is a direct mapped cache with only one entry per hash value; there are no hardware-walked overflow chains[4]. The hash function that is currently implemented by Itanium hardware is also very simple: it calculates the bitwise *exclusive OR* of the page number with the address space number. If address space numbers are allocated consecutively (which is the case for Linux) and page numbers co-incide between those consecutive address spaces, the result is frequent pathological collisions. In order to address this, vNUMA re-arranges the bit order of address space identifiers to improve the distribution of hash entries. This is not particularly elegant, however, in that it makes assumptions about operating system behaviour. If an operating system was to choose an inverse redistribution function (which it may well do in order to address the same problem), the pathological behaviour might re-emerge.

The fact that long-format entries are four words in size, compared to the single-word short-format entries, is a mixed blessing. This larger entry size is required to allow multiple address spaces to exist in the one page table, by tagging each entry with the address space it belongs to, as well as allowing usage of other Itanium TLB features such as protection keys and multiple page sizes (although neither of those features are currently used by vNUMA). It also provides ample space to store hypervisor metadata associated with a mapping, obviating the need for additional data structures. Specifically, while the hashed page table nominally tracks the *effective* mapping from virtual addresses to physical addresses, so that it can be used directly by processor hardware, the hypervisor may also need to determine the intermediate metaphysical address and OS-requested protections. This information can easily be stored in spare bits of the long-format page table

[4]There is a reserved field in each entry which could be used to add software-walked overflow chains, but then the additional entries are no longer transparently accessed by hardware, and the structure would no longer be naturally bounded in size. It would be preferable if Itanium supported multiple hardware-checked entries per hash value, as in the hashed page table defined by the PowerPC architecture [65].

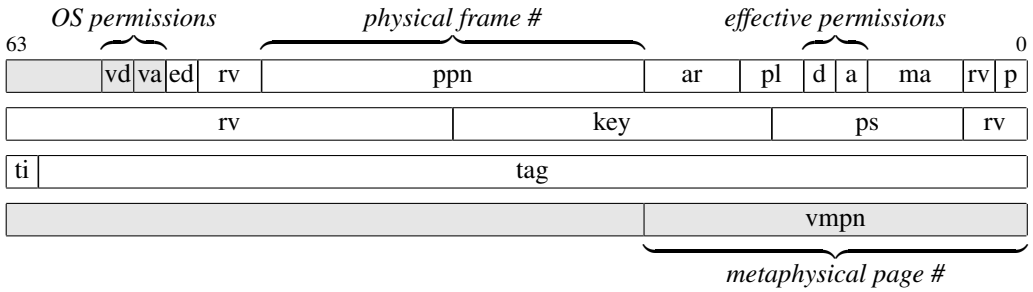



**Figure 5.5:** Anatomy of a long-format page table entry. Unshaded portions are part of the Itanium architectural definition [47], and more information can be found there; shaded portions are hardware-ignored and used by vNUMA.

entries, as shown in Figure 5.5. The disadvantage of the larger entry size is that it can impose greater demands on processor caches.

Even though vNUMA uses a fixed page size (usually 4 KiB) for its metaphysical-to-physical mappings, the operating system may insert virtual mappings of any page size. A naïve approach would insert a page table entry for every 4 KiB subpage of the large page, however this is clearly inefficient for very large page sizes. For example, Linux typically uses 16 MiB or 64 MiB pages for mapping physical memory. Inserting 4096 or 16384 consecutive 4 KiB mappings into the hashed page table would make the mapping operation very expensive and evict many existing hash table entries.

Thus, very large pages (>64 KiB) are instead inserted into a separate mapping directory, and then smaller subpages are faulted into the hashed page table on demand. (This mapping directory is also used for the pinned mappings referred to earlier; thus, if those mappings are ever evicted from the hashed page table, they can be restored on demand from the mapping directory.) With Linux as the guest OS, this mapping directory only grows to a handful of entries, so it is currently implemented as an array of eight linked lists — one per region of the address space — but it could easily be replaced by a better data structure if necessary.

However, even if large pages are not directly inserted into the hashed page table, purging large areas of the address space is still inefficient: the hypervisor must individually check each 4 KiB sub-region of the requested range in the

hashed page table to determine whether it needs invalidation[5]. Improving this is a difficult feat. One possibility is to handle large purges by purging an entire region (one-eighth of the address space), which can be done in O(1) time by changing the address space number. However this introduces a large indirect cost since many mappings may need to be re-established. If a bitmap was maintained describing which areas of the address space have been mapped, relevant parts of the purge range could more easily be targetted for shoot-down. Of course the Itanium address space is very big and an efficient hierarchical encoding for this bitmap would be needed. This could be the subject of future work.

It should also be noted that the implementation described violates the Itanium architecture in one respect. A TLB purge for a given address range is defined to flush all TLB entries that *overlap* with that range. This implies that inserting a large page and then purging a small sub-region should purge the entire large page. However, as a result of treating the large page as a collection of 4 KiB pages within the hashed page table, information is lost about the original mapping size that should be purged. Tagging the hashed page table entries with the insertion page size does not solve the problem, since the page table entries within the requested purge range may have been overwritten by hash collisions; there is no way for the hypervisor to know which entries outside the purge range must be checked. A practical solution is to perform purges at 64 KiB granularity, i.e. 16 entries at a time; then it is guaranteed that pages of 64 KiB and smaller will be flushed, and information about pages larger than 64 KiB can be found in the mapping directory. However, this imposes a slight performance overhead for little practical benefit: operating systems generally issue a purge for the entire range that they expect to be purged, and do not rely on the original mapping size. This is certainly true for Linux.

### Inverse mappings

The structures described so far — the frame table, the hashed page table and the mapping directory — enable the hypervisor to efficiently resolve translations from

[5] In fact, since the Itanium architecture defines that a TLB insertion purges previous translations for that virtual address range prior to inserting the new translation, this also has an impact on insertion performance.

higher to lower layers: from virtual to metaphysical, metaphysical to physical, and virtual to physical. It is also necessary to track the inverse mappings from metaphysical to virtual addresses, so that the relevant virtual mappings can be updated when the attributes of a metaphysical page are changed. This is a one-to-many mapping, as each metaphysical page can simultaneously be mapped at more than one virtual address (and in some cases *must* be: for example, some small application binaries have the one underlying page mapped as code at one virtual address and data at another virtual address, and both must be mapped simultaneously for forward progress).

In operating systems and some other hypervisors, the data structure that is often used for inverse mappings is a linked list of arrays, where each array has space for a certain number of mappings. However, these systems have the advantage that they can track the lifetime of address spaces, allocating and de-allocating these lists as necessary. Itanium address spaces have a very long lifetime: an operating system simply uses new address space numbers and leaves the old address spaces to naturally expire from the TLB. This makes it especially important that there is a size limitation on the number of mappings cached; an ideal data structure would make it easy to target the least-recently-used (or at least least-recently-inserted) entries for replacement.

A novel data structure, shown in Figure 5.6, was designed to satisfy these needs. The most recent mapping for any given metaphysical frame is contained in its frame table entry. Then, overflow chains are stored in a circular buffer, which makes allocation simple and provides a natural size limitation. To avoid having to explicitly remove entries from chains when they are overwritten (which would require doubly linked lists), the following trick is used. All of the pointers are monotonically increasing indexes; they are not modulo the buffer size (although the latter must be calculated when actually accessing the buffer). When traversing the overflow chains, instead of checking for a NULL pointer, one checks whether the index falls inside the current window of indexes that are still live in the buffer. Thus, pointers to entries that have been overwritten automatically fall out of scope[6]. Assuming that entries are always added to the beginning of a chain,

[6]It is still necessary to flush mappings from the TLB/VHPT when the corresponding inverse mappings are overwritten, to ensure that the set of inverse mappings is complete.

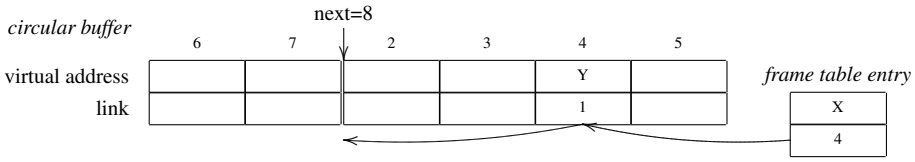



**Figure 5.6:** Tracking inverse mappings using a circular buffer as the overflow structure. In this example there are 6 slots in the buffer, and the next index to be written is 8 (which corresponds to slot 2); thus the currently valid indices are 2..7. Hence the link field with a value of 1 is not followed, as the entry has been overwritten.

all further entries must be even older, and so encountering an out-of-scope index terminates the list. In practice, since the indexes are of finite size, monotonicity will eventually be violated when the index wraps around; however, even for a 32-bit index this is very rare, and a complete flush of mappings can be done in this case.

### 5.2.3 Devices

The para-virtualised device and firmware interface from Linux-on-Linux was retained for the standalone version of vNUMA, but the backend implementation is more problematic since UNIX devices can no longer be used. A locally-developed device driver framework was employed. This was originally written for the Mungi operating system [58], but it has also been used to implement user-level drivers on Linux [27]; it is designed to be widely portable. Included as part of this framework are drivers for IDE and two popular Gigabit Ethernet chipsets (National Semiconductor DP83820 and Intel PRO/1000); the author later added another Gigabit Ethernet chipset (Broadcom Tigon3) and SCSI (LSI Logic Fusion-MPT). The Gigabit Ethernet drivers are critically important for implementing the distributed hypervisor described in Part II. Indeed the framework proved its portability; adapting it to the spartan environment of the vNUMA hypervisor was easy, and no doubt much easier than trying to extricate drivers from Linux or BSD. The disadvantage is that there are fewer drivers currently available for this framework, so this limits the hardware supported by the standalone version of vNUMA. In an ideal world, there would be a standard,

portable driver framework which all drivers are written to, maximising code reuse.

# Part II

# Transparent distribution

Having completed the first task of designing an efficient virtual machine monitor, it is now possible to address the core goal of vNUMA. That is, to build a virtual machine spanning multiple separate workstations, in order to combine their processing power into a single virtual NUMA computer.

There are two aspects to achieving this goal. Firstly, there is some infrastructure required to co-ordinate the separate workstations, send inter-processor messages and emulate devices. This enabling infrastructure will be described in Chapter 9. However, a far bigger challenge is the *distributed shared memory* (DSM) system, which is responsible for providing the illusion of a globally shared memory. Since memory is such a fundamental abstraction in computer systems, the performance of the DSM system is paramount.

There is already a large body of research on distributed shared memory. However, achieving good performance in vNUMA involves different goals and trade-offs. Most existing DSM systems have been designed as middleware for writing distributed computational applications, and as such they vary the programming model in order to achieve the desired balance between simplicity and performance. In contrast, vNUMA must faithfully reproduce the hardware SMP programming model so that unmodified applications and operating systems can be run. The shared address space in vNUMA is not just some subset of data memory that is known to be shared, but all of the memory of the virtual machine — including private data, instruction pages and register stack pages — thus efficiently and correctly handling a variety of access patterns is imperative.

However, there are also some unique opportunities for optimisation in vNUMA. Since vNUMA is a virtual machine monitor that runs in processor privileged mode, this paves the way for certain techniques that may be difficult or prohibitively inefficient in a userspace DSM implementation. For example, in certain cases vNUMA intercepts individual write instructions and emulates them (Section 8.2); in other cases it uses the processor's performance monitoring hardware to track the execution of certain instructions (Sections 7.5 and 8.8). These are novel optimisations that go beyond the techniques used in past DSM systems.

This part of the thesis will present an overview of distributed shared memory, the design of the vNUMA system, and the optimisations that were developed in order to improve the performance of the system.

# Chapter 6

# Shared memory

Multiprocessor programming models involve one or both of two paradigms: *message passing* and *shared memory*. In a message-passing model, the programmer must explicitly send and receive data in discrete packages. In a shared-memory model, communication occurs implicitly by writing data to certain memory locations that can then be read by other processors. Shared memory has long been considered simpler for the programmer, and for this reason it is the 'holy grail' of multiprocessor system design, despite the implementation challenges it presents.

The first generation of symmetric multiprocessors did not have caches and hence shared memory was a natural paradigm. All processors simply accessed the same banks of physical memory in an interleaved fashion. A write to a memory location would immediately be seen by a read from another processor in a subsequent bus cycle.

However, as processors became more powerful, local caches at each processor became essential for performance. In order to maintain the shared memory abstraction, sophisticated protocols called *cache coherency protocols* were now required to keep the data in the separate caches synchronised.

In small systems with a single memory bus, this can be achieved by having each processor monitor the transactions on the memory bus and either update its own cache with new data or simply invalidate any conflicting cache lines: this are known as a *snoopy* protocol. A single shared bus is impractical in larger systems, so beyond the local bus these snoopy protocols can be supplemented

with *directory-based* protocols, wherein a central directory tracks which groups of processors have possession or ownership of a particular cache line. The simpler single-bus topology is described as *symmetric multiprocessing* (SMP), reflecting the fact that all processors experience the same latency accessing memory. Larger multi-bus systems are described as *non-uniform memory access* (NUMA). NUMA systems that employ cache coherency protocols to maintain transparency of shared memory access — the vast majority of commercial NUMA systems — are sometimes further labelled as *cache-coherent non-uniform memory access* (ccNUMA) systems.

It has also been demonstrated that similar coherency protocols can be implemented in software, without special hardware support for shared memory. This can be achieved either by instrumenting memory accesses or by using virtual memory techniques to transparently intercept memory accesses. This allows a shared memory programming model to be used on networks of general purpose computers that do not have any hardware mechanism for cache coherency. Such systems are known as distributed shared memory (DSM) systems.

## 6.1 Early software DSM systems

The ancestor of most modern DSM systems is IVY (Integrated shared Virtual memory at Yale), developed in the late 1980s by Kai Li and Paul Hudak at Yale University [61]. IVY was the first system to implement software distributed shared memory by varying protections on virtual memory pages, which allowed the shared memory to be simulated transparently to the application program. The basic concepts developed by Li and Hudak are widely used in later DSM systems.

The Li-Hudak DSM algorithm can be characterised as a multiple-reader/single-writer protocol. At any point in time, either multiple nodes have read-only replicas of a page, or a single node has the one writable copy and no other nodes have access. If the page is in the read-shared mode and a node requires write access, it must first request other nodes to invalidate their copies of the page. Execution only continues after replies have been received, guaranteeing that only one node has write access to a page at any one time. Thus there are never any conflicts

between writes simultaneously made by different nodes, because no page is ever simultaneously writable on more than one node.

This can be made transparent to the application program by using the virtual memory system of modern processors. For example, when a page is read-shared, the page is made read-only to the application. If the application writes to the page, a virtual memory fault occurs and the DSM system takes control, requesting other nodes to invalidate their copies. Once this is complete, the page is made writable and the application is continued.

Clearly an essential part of such a scheme is being able to track which nodes have copies of the page (the *copyset*), and which node has the single authoritative copy (the *owner*). A node that performs the function of tracking the owner is known as the *manager* of a page. Li and Hudak experimented with several different schemes:

- In a **centralised** manager scheme, there is a single manager on the network which tracks ownership for all pages. Clearly, this manager can easily become a bottleneck on the network, hindering scalability.

- In a **fixed distributed** manager scheme, the set of pages is statically partitioned between different managers, such that there is no single bottleneck. The manager for a page always knows the owner, so the owner of a page can always be reached in a maximum of two hops; this is reduced to one hop if either the faulting node is the manager or the manager is the current owner. Unless the partition function is chosen with knowledge of the application, probability will tend to favour the two-hop case on larger clusters.

- In a **dynamic distributed** manager scheme, there is no fixed manager for a page; rather each node is responsible for remembering the last known owner, and messages are forwarded until the current owner is reached. This scheme can improve performance for certain page access patterns — only one hop is required if the last known owner is indeed still the owner of the page — but also increases the potential worst-case cost. Intermittent broadcasts may be used to more aggressively notify nodes of ownership changes.

Other early DSM systems that followed IVY include those in Mirage [31]. The Mirage DSM system is similar to that in IVY, but was implemented in an operating system kernel rather than in userspace, allowing even greater transparency: distributed shared memory segments could be created in the same way as local interprocess shared memory segments. As well as introducing small optimisations to the IVY protocol, Mirage's major contribution was an attempt to address the page thrashing problem, which will be discussed further in Section 7.5. Page thrashing occurs when two or more nodes simultaneously demand write access to the same page; depending on network timing, ownership may be continuously transferred between the two nodes with neither performing useful work. Mirage attempted to address this by guaranteeing that each recipient of a page retains access for a certain minimum period of time, hopefully allowing it to make progress.

Clouds [19] was designed from the ground up as a novel distributed object-oriented operating system; the goal of its DSM layer is to permit network-transparent access to objects in the system. The DSM protocol is somewhat simpler than those implemented in IVY and Mirage, utilising a centralised manager executing on a separate UNIX computer. However, Clouds also attempted to address the thrashing problem (as well as providing richer synchronisation options) by introducing a concept of locks on memory segments, so that the application can lock an area of memory either exclusively (write) or shared (read). This potentially provides better performance than the time-based approach of Mirage, but at the expense of transparency; an application must be specially written to take advantage of these locks.

There are a number of other similar first-generation DSM systems in the literature, which are not necessary to describe in detail here (a summary is provided in [77]). Before describing later DSM systems, the notion of consistency models should be defined.

## 6.2 Consistency models

Distributed shared memory systems can be characterised by the *consistency model* that they provide. Fundamentally, a program interacts with the memory system

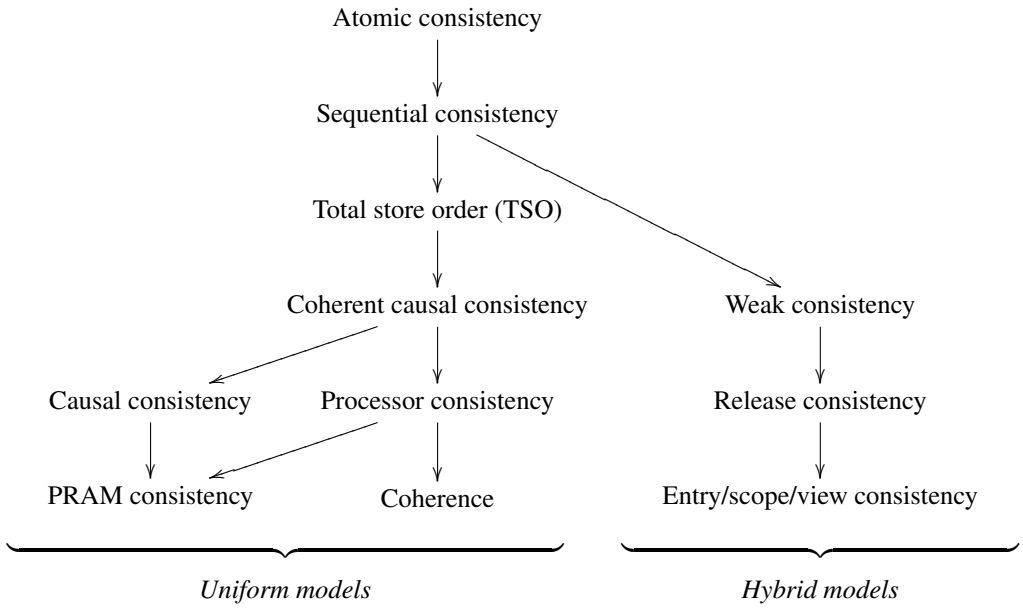



**Figure 6.1:** Hierarchy of common consistency models

by issuing writes and reads. The consistency model restricts the set of possible outcomes for the reads, dictating which of the written values is returned by each read. On a single processor, one expects a read to observe the value written by the most recent write; however in a distributed system there may be various useful interpretations of "most recent".

One can arrange consistency models into a hierarchy. An overview of the relationship between a number of common consistency models is provided in Figure 6.1, loosely based on a similar diagram presented by Mosberger [72]. A program that is correctly specified for a more relaxed consistency model — one that is lower in the tree — will always execute correctly on a stricter consistency model (the reverse is not true). However, stricter consistency models come at a cost in performance; relaxed consistency models generally provide greater scope for optimisations to be made in the shared memory system.

The models in the figure will be described shortly, but first some terminology must be defined. In describing consistency models one frequently refers to *observation* and *visibility* of accesses. Observation relates to a specific pair of

accesses: a write is observed by a read when the read returns the value written; a write is observed by a processor when a read on that processor observes the write. In contrast, visibility is defined in terms of the outcome of a hypothetical access: if a write on processor P1 can at some later time be observed by a hypothetical read on processor P2, that write can be said to have become visible to P2[1]. Read visibility also can be defined: if a hypothetical write on processor P2 can no longer affect the value returned by a read on P1, then that read has become visible to P2 [24][2]. Visibility can be a useful concept since it allows discussion of certain dataflow properties without considering a specific execution.

In this section, brief informal summaries of common consistency models are provided. More formal definitions can be found in many places such as [81,39,20] and papers cited in individual definitions. Note that there is some confusion in the field regarding consistency models and definitions; however, based on a wide survey of literature, the author believes that the following definitions represent the most common usage.

### 6.2.1 Atomic consistency (Linearisability)

Atomic consistency is the strongest possible consistency model. An atomically consistent system produces the same result as if all the operations were executed serially, in the specific real-time order in which they actually occurred[3]. Thus, the notion of reads returning the "most recently" written value has its most intuitive meaning: if a read on any processor occurs time-wise after the completion of a write on any processor, then the read must return the data written. From the point of view of write visibility, this requires that writes become visible to all processors immediately after completion. Clearly, instantaneous notification is impossible in practice, so this implies that a write must always invalidate cached values held by other processors *before* it can be allowed to complete. The Li-Hudak protocol

[1] This does not necessarily imply that P2 already has a read copy; in an invalidation-based protocol, the hypothetical read may need to fetch the value.

[2] If one assumes that the value returned by a read can no longer be affected after the read completes (for some architecture-specific definition thereof), this implies that a read normally becomes visible everywhere at some time before it completes.

[3] Some definitions of atomic consistency allow that, in the case that operations overlap and hence there is no clear real-time ordering, sequential consistency applies.

described in the preceding section is an example of such a protocol and indeed it satisfies atomic consistency.

### 6.2.2 Sequential consistency (Sequentialisability)

A related fundamental consistency model is sequential consistency. As defined by Lamport [56], it requires that the outcome of a parallel execution is one that is possible to achieve via a serial execution in which the threads are interleaved — but not necessarily one that corresponds to the original real-time order, as is the case for atomic consistency. It must be possible to somehow merge the set of operations that occurred into some global order that preserves program order and the property known as *legality*: that there are no conflicting writes between a read and the write that supplied the corresponding data.

Unfortunately, this definition does not immediately suggest a method of enforcement, and indeed the problem of determining whether a particular execution obeys sequential consistency is believed to be NP-complete [35]. Real systems guaranteeing sequential consistency do so by imposing an easily enforceable constraint on the execution. Constraints that have been shown to guarantee sequential consistency include: [70]

- *Write-write (WW) constraint*: The system enforces that all writes become visible to all processors in the same order.
- *Object-ordered (OO) constraint*: The system enforces that any pair of *conflicting* accesses appear to occur in the same order to all processors. Conflicting accesses are any pair of accesses to a single object or data item, in which at least one is a write (i.e. write-write, read-write and write-read ordering is necessary; only read-read re-ordering is allowed).

### 6.2.3 Total store order (TSO)

Total store order is a relaxation of sequential consistency which has applications to many real processor architectures. TSO is defined similarly to sequential consistency, but allows a weakening of the program order restriction: earlier writes and later reads on the same processor need not have this specific order

preserved in the total order. In practice, this allows for a processor to buffer writes and satisfy reads using those pending writes, before those writes become visible globally. Thus, there exists a global ordering of writes, but any particular processor can observe its own writes early.

### 6.2.4 PRAM consistency (FIFO consistency)

PRAM (pipelined RAM) consistency is weaker and merely requires that writes of a given processor become visible in the order in which they were issued. In other words, any two writes made by the same processor are observed in that specific order, but different processors' operations may be differently interleaved to various observers; there is no global ordering of writes.

### 6.2.5 Causal consistency

Causal consistency strengthens PRAM consistency to take into account not only writes by a single processor but also transitive causal relationships. For example, consider if processor P1 issues a write $X = 1$, and then a read on P2 observes $X = 1$ causing it to issue a write $X = 2$. Causal consistency will guarantee that another processor P3 will observe $X = 1$ before $X = 2$, whereas PRAM consistency would allow either order. While this is still weaker than sequential consistency and TSO, it satisfies common programming assumptions about causality, and the majority of programs execute unmodified with causal consistency [3].

### 6.2.6 Coherence

Whereas the previous consistency models apply to the whole of memory, coherence is defined for finer granularity objects — for example individual memory locations, cache lines or pages. Technically, coherence specifies that writes *to a single object* must become visible to all processors in the same order, although most practical implementations allow a writer to observe its own writes early as in TSO consistency. Thus coherence is sequential consistency or TSO consistency applied at object granularity. One can also combine coherence with the weaker models, resulting in:

- **Processor consistency = Coherence + PRAM consistency:**
  Processor consistency combines coherence on individual objects with PRAM consistency for the entire address space.

- **Coherent causal consistency = Coherence + Causal consistency:**
  Coherent causal consistency combines coherence on individual objects with causal consistency for the entire address space.

### 6.2.7 Weak consistency

All of the previous consistency models are referred to as *uniform* models, as all operations are treated equally. In contrast, weak consistency is a *hybrid* model which provides the programmer with two different types of operations, *synchronisation* operations and *unordered* operations. Synchronisation operations are guaranteed to obey a strong consistency model, such as sequential consistency. Unordered accesses are only ordered with respect to the synchronisation operations, and may be arbitrarily re-ordered amongst themselves (subject to data-flow considerations). Assuming that the majority of program operations are unordered, the overhead of maintaining consistency can be reduced.

### 6.2.8 Release consistency

Release consistency [34] enhances weak consistency by providing two different types of synchronisation operations, *acquire* and *release*. This terminology refers to their association with critical section semantics. An acquire is used at the start of a critical section, and prevents unordered accesses following the acquire from being ordered before the acquire. A release is used at the end of a critical section, and prevents unordered accesses preceding the release from being ordered after the release. In effect, accesses are prevented from "leaking out" of the enclosing critical section; but unlike weak consistency the ordering is one-way and does not impose restrictions on unordered accesses outside the critical section. There may also be a *barrier* or *fence* operation which prevents unordered accesses from crossing it in both directions.

### 6.2.9 Entry, scope and view-based consistency

These consistency models are based on release consistency, but add the concept that each critical section only 'protects' a certain set of shared objects. Thus, they bind each lock to specific shared objects; only those objects are guaranteed to be made consistent by the acquire/release mechanism, instead of the entire address space. This reduces the set of writes or invalidations that need to be communicated at acquire/release time.

In entry consistency [7], the programmer must explicitly make the bindings between locks and objects; clearly, this places a large burden on the programmer and is error-prone. Scope consistency [43] and view-based consistency [41] improve on entry consistency by establishing those bindings dynamically, at least for certain classes of programs that include appropriate synchronisation operations.

## 6.3 Later software DSM systems

All of the DSM systems described so far — IVY, Mirage and Clouds — implement atomic consistency. More recently, DSM research has focused on more relaxed consistency models, specifically release consistency, entry consistency, scope consistency and view-based consistency.

Munin [9] was the first system to leverage release consistency to allow multiple simultaneous writers. Rather than synchronously invalidating all other copies when a write occurs, Munin allowed writes to be accumulated locally until a synchronisation event forced those writes to be propagated to other nodes. In Munin, writes were propagated at the time of a release, which is known as an *eager release consistency* protocol. TreadMarks [50], one of the most commercially successful DSM systems, pioneered the idea of *lazy release consistency* whereby the propagation of updates is delayed until the next acquire by another node; this can reduce the number of messages, at the expense of complicated bookkeeping [51]. Some newer systems also use a hybrid scheme, *home-based lazy release consistency*, which propagates updates eagerly to a home node, whereupon the

| Architecture | Base model | Per-location | Store atomicity | Atomic RW | Fences |
|---|---|---|---|---|---|
| MIPS | Sequential | Sequential | ✓ | ✓ | ✗ |
| SPARC TSO | TSO | TSO | ✓ | ✓ | ✓ |
| Itanium strong[a] | TSO | TSO | ✓ | ✓ | ✓ |
| IA-32 | Processor | TSO | ✗ | ✓ | ✓ |
| Alpha | Weak | TSO | ✓ | ✓ | ✓ |
| SPARC RMO | Weak | TSOrr[b] | ✓ | ✓ | ✓ |
| PowerPC | Weak | TSOrr | ✗ | ✓ | ✓ |
| Itanium weak | Release | TSOrr | ✓/✗[c] | ✓ | ✓ |

[a] When ordered forms of load and store instructions are used.
[b] TSOrr is a read relaxation of TSO; independent reads may be performed out of order.
[c] Stores with release annotations become visible atomically, ordinary stores do not.

**Table 6.1:** Memory consistency models of common processor architectures

updates are discarded; other nodes then lazily fetch the new page data from the home node [44].

Aside from release consistency, other systems have also implemented entry consistency (Midway [7]), scope consistency (JIAJIA [26], Brazos [85]) and view-based consistency (VODCA [40]). As described in the previous section, these further relax the consistency model by associating specific objects with critical sections.

## 6.4 Hardware shared-memory systems

In contrast, when building a system such as vNUMA — with a goal to execute legacy software designed for hardware shared-memory systems — one does not have the luxury of choosing an arbitrary consistency model. In order for the legacy software to function correctly, it is necessary to provide the same memory consistency guarantees as the hardware system originally provided to the programmer.

To this end, the programmer-visible consistency models of several commercial architectures are summarised in Table 6.1 (in general, this applies to both SMP and ccNUMA implementations of each architecture). Itanium, which is of primary importance to this thesis, has been considered here in the context of two operating modes: a strongly-ordered mode in which all loads and stores are of the

ordered form (as used by the IA-32 emulation mode, or by a paranoid Itanium programmer), and a weakly-ordered mode in which ordinary loads and stores are the norm. SPARC can also operate in a number of different ordering modes; the table considers the most commonly used modes, namely *total store order* (TSO) and *relaxed memory ordering* (RMO).

Out of these, only MIPS processors attempt to provide true sequential consistency without the need for fences[4]. SPARC and Itanium's strong models both provide TSO, which is a little weaker than sequential consistency in that reads may be satisfied from local write buffers before those writes become remotely visible. IA-32 provides processor consistency, which guarantees that the writes of a given processor become visible in order, but does not provide a global total ordering or causality guarantees. Alpha, PowerPC, SPARC and Itanium's models are weaker still; the majority of operations can be arbitrarily re-ordered, and fence instructions must be inserted where this re-ordering is to be prevented. Itanium goes further in that it provides uni-directional half-fences labelled acquire and release, motivated by the ideas of release consistency.

However, even these weaker models are surprisingly strong; certainly stronger than a typical software DSM implementation making use of a relaxed consistency model. The first striking difference is that all of the models guarantee coherence: accesses to a single location must obey TSO (or at worst, TSOrr, which is similar to TSO but does not guarantee the fulfilment order of two consecutive reads at an observer). In contrast, the software implementations described in the previous section do not provide any ordering guarantees for unordered writes to the same location; generally this would be considered a data race and forbidden.

All of the surveyed architectures also provide atomic *read-modify-write* instructions that expect to access the shared memory atomically. All except MIPS provide *fence* operations that can be used to restrict ordering and ultimately, with the right combination of fences, achieve execution compatible with sequential consistency. While software DSM systems also provide synchronisation primitives such as locks and barriers, their semantics are often quite different.

[4]Note that in order to achieve reasonable performance, advanced implementations such as R10000 actually issue some operations speculatively and then roll back the program if sequential consistency requirements are not met.

Finally, many of the models require *remote store atomicity*: the property that, once a write becomes visible to a processor other than the writer, it must also atomically become visible to all other processors. This also implies that there must exist a total store ordering, even if that ordering is not perceivable by individual processors. Amongst the architectures surveyed, this is explicitly required by MIPS, SPARC and Alpha, but not by IA-32 and PowerPC. In the case of Itanium, it is only required for stores with release annotations but not for ordinary stores; realistically this is of little comfort to a software DSM designer, since an annotated store can occur at any time and the infrastructure for remote store atomicity must be in place.

Thus, even the memory models that ostensibly provide weak consistency are stricter than desirable for a software DSM system. They permit a great deal of local re-ordering, but there is a pervading assumption of an underlying total store order provided by the interconnect. Current DSM systems implementing release consistency cannot cope with such requirements; they are unable to guarantee per-location coherence, or store atomicity, or indeed correct execution of atomic operations on the shared memory. They also assume that acquires and releases come in pairs, which is not the case in the Itanium architecture; Itanium acquires and releases are simply half-fence operations which can occur arbitrarily often.

The challenge facing vNUMA, which will be addressed in this thesis, is to provide a software DSM system which can satisfy these requirements while simultaneously taking advantage of optimisation opportunities. Just like there are atomically-consistent cache consistency protocols underlying SMP consistency models, so the base vNUMA protocol is atomically consistent; this base protocol will be described in the next section. The other vNUMA-like systems that were described in Chapter 1 only use such basic protocols. In vNUMA, this is enhanced with more advanced write-update schemes that improve performance while maintaining a consistency model compatible with TSO (Section 8.1). Finally, the weakness of the Itanium model is exploited to allow some degree of local re-ordering (Section 8.8).

# Chapter 7

# The vNUMA DSM system

## 7.1 Integration with hypervisor

The most obvious novelty of vNUMA lies in the fact that the distributed shared memory system is integrated at the hypervisor layer, which allows shared memory to be provided completely transparently to a legacy operating system.

The hypervisor memory management model was detailed in Section 5.2.2, and is summarised by the three layers depicted in Figure 7.1: virtual addressing, metaphysical addressing and physical addressing. Applications generally execute in a virtual address space. The operating system manages metaphysical addresses, which embody the linear memory space of the virtual machine, and establishes mappings from virtual addresses to metaphysical addresses. These mappings can include permission information ($p_v$) specifying whether read and/or write access is allowed through these mappings. Then, the hypervisor is responsible for mapping

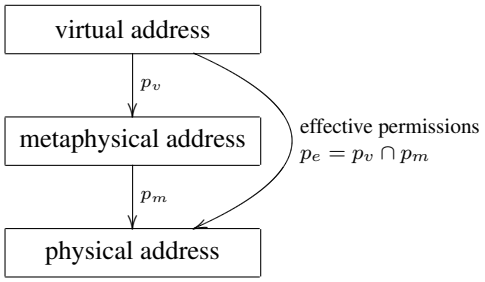



**Figure 7.1:** Address translation layers

metaphysical addresses to physical addresses, representing the RAM of the real computer hosting the virtual machine.

To allow applications to execute transparently, the hypervisor establishes mappings in the real processor's TLB (and the architectural pagetable: the VHPT) that correspond to the composition of the two mappings, i.e. mapping directly from virtual to physical addresses. The hypervisor may also insert mappings corresponding to the metaphysical to physical mapping, to provide a fake physical address space for the guest operating system. In no case does the guest operating system execute in the processor's real physical mode, which would expose it to the real physical address space.

In vNUMA, the operating system sees a single global metaphysical address space, regardless of which node it executes on. All instructions and data which can be accessed by the operating system and applications exist within that shared metaphysical address space. Either the operating system can opt to access metaphysical addresses directly, or it can establish virtual mappings to metaphysical pages on a per-processor basis, and then access data through virtual addresses. From the perspective of the operating system, this is indistinguishable from a real NUMA system.

This illusion of a single global metaphysical address space is achieved by selectively restricting the metaphysical page permissions — $p_m$ in Figure 7.1 — according to a DSM algorithm, and performing any necessary remote synchronisation operations when a protection fault occurs. This is exactly analogous to the way in which a userspace DSM system implements a DSM algorithm by restricting virtual page permissions. In the case of vNUMA, accesses may be made through both metaphysical and virtual mappings, but the hypervisor ensures that permissions on virtual mappings are constrained by the permissions on the underlying metaphysical mappings, i.e. $p_e = p_v \cap p_m$. Thus, regardless of whether a prohibited access occurs through virtual addressing or metaphysical addressing, a page fault occurs and the DSM system is invoked.

Crucially, the hypervisor's memory management layer must provide the ability to update the DSM-specified permissions $p_m$ at will, which involves updating not only the metaphysical-to-physical mapping but also any virtual-to-physical mappings that have been composed from it. Generally this implies that it must

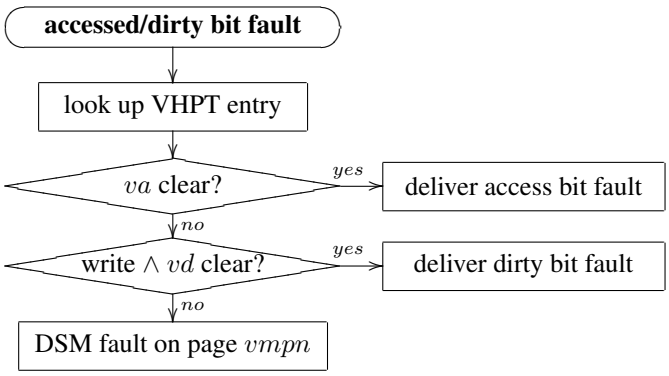



**Figure 7.2:** Flowchart of protection fault handling

maintain reverse mappings from each metaphysical page to the set of virtual addresses where it is mapped, so that each virtual mapping can be updated. An efficient scheme for this was presented as part of the vNUMA memory management design in Section 5.2.2.

In practice, the DSM permissions are enforced using the accessed ($a$) and dirty ($d$) bits in the Itanium TLB and VHPT. Clearing the accessed bit causes a fault on both read and write access to a page; clearing the dirty bit causes a fault on write access only[1]. Note that the operating system may also use the $a$/$d$ bits for its own protection needs; these OS-specified permission bits will be referred to here as $va$/$vd$. The effective values of $a$/$d$ are calculated as the intersection (specifically, logical AND) of the DSM-specified permissions and the OS-specified permissions.

A flowchart for fault handling is shown in Figure 7.2. The fault handler inspects the pagetable (VHPT) entry corresponding to the fault address. The structure of a VHPT entry is shown in Figure 7.3. While nominally the VHPT entry maps a virtual page directly to a physical frame, vNUMA also maintains — within the hardware-ignored portion of the entry — information about the original mapping made by the operating system, namely the metaphysical page number ($vmpn$) and any OS-imposed protections ($va$, $vd$).

Using this information, the fault handler first checks whether the fault was

[1]The access rights ($ar$) field also provides a separate protection mechanism, but the $a$/$d$ mechanism is simpler and sufficient for the requirements of the DSM algorithm.

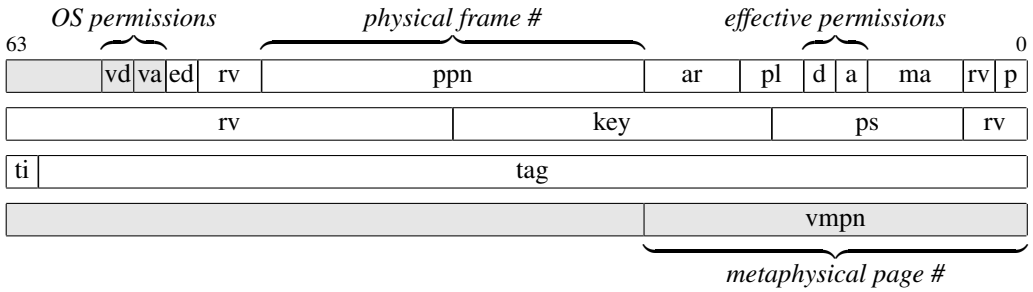



**Figure 7.3:** Anatomy of a VHPT entry. Unshaded portions are part of the Itanium architectural definition [47], and more information can be found there; shaded portions are hardware-ignored and used by vNUMA.

due to those OS-specified protections, in which case the fault is reflected to the operating system. If not, then the fault must be due to DSM-specified protections, and the DSM system is invoked using the metaphysical page number from the VHPT entry. All DSM requests ultimately use metaphysical addresses.

## 7.2 Basic protocol

At the heart of the vNUMA DSM system is a simple atomically-consistent single-writer/multiple-reader protocol. This core protocol, based on the Li-Hudak protocol that was described in Section 6.1, provides a solid and reliable foundation to enable correct execution of all existing SMP programs (and the Linux operating system itself), since atomic consistency provides consistency guarantees at least as strong as those provided by SMP and NUMA systems. It also provides a baseline for performance and correctness comparisons.

There are a number of choices to be made when implementing such a protocol. For page location, vNUMA implements a fixed distributed manager scheme, whereby the global metaphysical address space is divided into equal-size portions; each node is responsible for managing one of those portions. Hence there is a simple fixed mapping from any given address to the manager node. In the ideal case, the manager is the node which accesses the page most often, since it always knows exactly where the page is located; this is normally the motivation behind a dynamic distributed manager scheme. However, in vNUMA, the operating

system is made aware of the locally-managed memory area via the concept of NUMA node-local memory, and will favour that memory when making allocation decisions. Thus, the fixed distributed manager scheme interacts well with a virtual NUMA system.

Further there are two variations of the fixed distributed manager algorithm described by Li and Hudak in their seminal paper [61]: an initial algorithm in which both the copyset and owner information is maintained by the manager, and an 'improved' algorithm in which the copyset is maintained by the owner and transferred with the page. From the point of view of a virtual NUMA system there may be some advantages to having the copyset available on the manager, particularly with respect to propagating updates in a write-update version of the protocol. However, that scheme increases the number of messages, and results in deadlock cases that are difficult to avoid (a formal analysis of the deadlocks involved in the Li-Hudak algorithms, using a model checker, was carried out by a group at the University of Utah [37]). For these reasons, the vNUMA protocol is based on the 'improved' variety of the algorithm.

There are some further refinements made to this algorithm, which are described in the following sections. Unlike the write-update scheme described later in this thesis, none of these optimisations have any effect on the program-visible consistency model; they are purely performance optimisations. Pseudocode for the overall algorithm is provided in Section 7.6.

## 7.3 Double fault optimisation

A *double fault* occurs when a node encounters a read fault on a page, obtains a read copy, and then encounters a write fault. In the naïve version of the Ivy DSM algorithm, the page data would be re-fetched on the write fault. This is clearly wasteful in the common case, since the node already has a read copy. However, if a third node simultaneously writes to the page, then it is not correct for both nodes to proceed using their cached data; one node proceeds first and then the other node must receive a modified version of the page data. Thus, a write request may conditionally return a copy of the page, and there is a question of how a responder decides whether or not to send the page data.

One common solution, presented as the "shrewd algorithm" in [52], is to maintain a version number for each page. This page version number is transferred along with the page and incremented each time read-write access is obtained. When sending a write request, a requester sends the version number of the page that it is in possession of. When the request is ultimately satisfied, the responder can check the version number in the request against the version number on its copy of the page, to decide whether or not fresh page data needs to be sent.

However, in actual fact this version number scheme is redundant; there is a simpler solution that is equivalent and does not require maintaining version numbers. The owner can simply inspect the copyset to decide whether or not new page data needs to be sent to a given destination node. If some third node happens to obtain write access first, its first action must be to invalidate all other nodes' read copies and remove those nodes from the copyset. Thus if the destination node is still in the copyset, that proves that there were no intervening writes and its read copy is still valid. If the destination node is not in the copyset, there must have been intervening writes and a new copy of the page must be sent. This method also seems to be used in Mirage [31].

## 7.4 Manager as preferred owner optimisation

The second minor optimisation to the base DSM algorithm involves the situation where a page is read-shared, and the manager node becomes part of the copyset but would not normally become the owner. Since further requests for the page, from other nodes, will be directed to the manager in the first instance, it would be desirable to be able to satisfy them at the manager. Clearly the manager already has a read replica so it could provide page data; however it does not have authority to maintain the metadata since it is not the owner.

The Mirage system addressed this issue by sending an advisory message from the manager to the owner so that the owner can update its metadata. However, vNUMA implements it in a simpler way which does not require the extra message: ownership is implicitly migrated to the manager when the manager becomes part of the copyset. That is, if a non-manager node is the owner and it receives a read

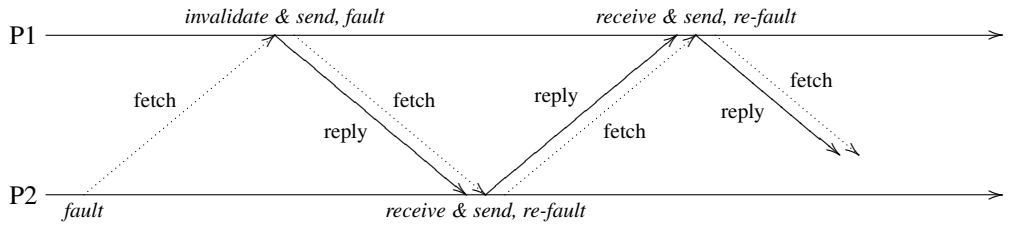



**Figure 7.4:** Timeline demonstrating the page thrashing problem. Solid lines indicate transfers of ownership.

request from the manager, it transfers ownership with the read reply. Thus, the manager becomes the owner and can service future requests locally.

## 7.5 Thrashing avoidance

A naïve DSM implementation suffers from the problem illustrated in Figure 7.4, often referred to as page thrashing. If two nodes simultaneously demand write access to a page, the page may be transferred back and forth with no useful work achieved. Assume that initially P2 sends a fetch message to P1 requesting ownership of the page. As soon as P1 invalidates its local copy and sends a fetch reply, it finds that it requires the page again, and sends a fetch message immediately after the fetch reply. P2 now receives a fetch reply and fetch back-to-back; assuming that it processes the fetch immediately and sends the page back to P1, a situation can occur where neither node makes any progress. Such a scenario is especially likely with fast networks, where the inter-packet delays are insufficient to break the resulting livelock.

One obvious solution, used in many DSM systems including Mirage, is to artificially introduce a delay to break the livelock. Depending on the protocol, this can either happen on the requester, whereby a node that is repeatedly faulting backs off and delays before trying again, or on the ‘responder’, whereby a node

delays passing on a newly received page (either by responding with a NAK and making the sender try again, or by simply delaying its response).

The problem with this timed backoff approach is that it is non-optimal by design. There is no easy way to determine the appropriate delay (except perhaps via a complex feedback algorithm). If the delay is too short, the livelock may not be broken. If the delay is too long, then it may waste valuable execution time.

For the vNUMA system, a more deterministic approach was desired, which would ensure that at least one instruction has executed each time a page is transferred. In this way, forward progress of the system can always be guaranteed.

In the initial implementation, this was done by putting the virtual machine into single step mode after receipt of a page. In the Itanium architecture, this means that a trap is raised after an instruction completes. Any remote requests for the page during this interim period are queued, and only serviced once the single step trap is received and hence it is known that progress was made.

A refinement of this idea, which does not require the single step trap, was implemented in a later version. In this improved version, a performance monitoring unit counter is used to count completed instructions; by checking this counter, vNUMA can always determine if progress has been made since a page was last received[2]. The advantage of this approach is that it incurs minimal overhead for pages that are not contended. If a lack of forward progress is detected, there are two options: one is to arrange for a single-step trap to be delivered once progress has been made, as in the previous approach; the other is to recheck for progress after a short time. The latter option was eventually chosen, since in the optimistic case this time window can allow more than one access to a page to complete before the page is relinquished. However, unlike the timed backoff scheme used in other DSM systems, this time delay can be set very short to minimise latency, since progress is guaranteed independently of it.

A complication with enforcing forward progress is that it introduces the possibility of deadlock. A single Itanium instruction can access up to four pages: an instruction page, a data page and two register stack engine pages[3]. In any case

[2]Note that it is not sufficient to check the instruction pointer to determine if progress has occurred, since code may be executing in a loop.

[3]This can occur when a memory access instruction is at the target of a return instruction, and the register stack engine crosses a page boundary while reloading registers for the procedure being

where pages are being held waiting for other pages, there is a possibility of a circular wait condition, in other words a classic deadlock.

For example, consider if processor P1 has just received page $A$ and is thus refusing to relinquish it until progress is made; P2 has just received page $B$ and is similarly refusing to relinquish it until progress is made. However, if P1 also requires page $B$ to make progress, and P2 also requires page $A$ to make progress, then a deadlock occurs. Naturally more complicated circular dependencies are possible.

In order to avoid such a deadlock, the livelock avoidance algorithm is only applied to pages accessed via explicit data references, and not instruction or register stack pages. Since the data reference is always logically the last reference made by an instruction — occuring after the instruction reference, and after any register stack accesses — instruction completion is guaranteed once the data page is obtained, and there is no possibility of deadlock. Indeed it is not necessary to apply the livelock prevention algorithm for instruction and register stack references, since instruction accesses are always reads, and register stack pages should not be simultaneously accessed by multiple CPUs (or undefined processor behaviour could result). Even if a malicious application were to invoke this livelock case, it would not prevent the operating system from taking control and the process could be killed. Thus, this strategy prevents livelock in a well-behaved operating system while also avoiding any possibility of deadlock.

On some other architectures such as x86, this approach might still result in deadlock, since a single instruction may access several data pages. One possibility would be to release pages after a random period of time, even if no progress is made. In the worst case, this re-introduces the problems associated with backoff algorithms, but should perform better in the common case, while ensuring that a permanent deadlock does not occur.

---

returned to.

## 7.6 Pseudocode

This section lists pseudocode for the core vNUMA DSM system. It is hoped that this will be a useful resource for future developers of DSM systems, since the pseudocode provided in many DSM papers is inadequate for practical implementation, and in some cases missing critical details. The pseudocode provided here is intended to reflect working source code as accurately as possible, including such details as the parameters used in messages, while omitting syntax details that would compromise readability.

When a program access causes a page fault that invokes the DSM system, the **fault** handler shown in the pseudocode is invoked, specifying the page and whether write access is required. This may send either **fetch** or **invalidate** messages, which eventually result in **fetch_reply** or **invalidate_reply** response messages. The handlers for all of these message types are also shown in the pseudocode. Several items of metadata are tracked on each node for each page: the owner of the page ($page.owner$), the copyset ($page.copyset$), the modes of access currently allowed on the local node ($page.permissions$), a lock for the page ($page.lock$) and a corresponding queue of requests waiting for the lock ($page.wait\_queue$).

**fault**($page$, $is\_write$): handles a page fault on a DSM page

$page.lock = 1$
**if** $page.owner = this\_node$ **then**
  send **invalidate**($this\_node$, $page$) to nodes in $page.copyset$
**else if** $manager(page) = this\_node$ **then**
  send **fetch**($this\_node$, $page$, $is\_write$) to $page.owner$
**else**
  send **fetch**($this\_node$, $page$, $is\_write$) to $manager(page)$
**end if**
wait while $page.lock = 1$

**fetch**($requesting\_node$, $page$, $is\_write$): server for **fetch** messages

**if** $page.lock$ **then**
  enqueue message on $page.wait\_queue$
**else if** no forward progress since $page$ received **then**
  enqueue message on $page.wait\_queue$
  schedule callback to process $page.wait\_queue$
**else if** $page.owner = this\_node$ **then**
  **if** $requesting\_node \notin page.copyset$ **then**
    send page data to $requesting\_node$
  **end if**
  send **fetch_reply**($this\_node$, $page$, $is\_write$, $page.copyset$) to $requesting\_node$

  **if** $is\_write$ **or** $manager(page) = requesting\_node$ **then**
    $page.owner = requesting\_node$
  **else**
    $page.copyset = page.copyset \cup \{requesting\_node\}$
  **end if**
  $page.permissions = is\_write$ ? nil : read
**else**
  send **fetch**($requesting\_node$, $page$, $is\_write$) to $page.owner$
**end if**

**fetch_reply**($from\_node$, $page$, $is\_write$, $copyset$): server for **fetch_reply**

**if** $is\_write$ **then**
  $page.owner = this\_node$
  $page.copyset = copyset \setminus \{this\_node\}$
  **if** $page.copyset \neq \emptyset$ **then**
    send **invalidate**($this\_node$, $page$) to nodes in $page.copyset$
    **return**
  **end if**
**else if** $manager(page) = this\_node$ **then**
  $page.owner = this\_node$
  $page.copyset = (copyset \setminus \{this\_node\}) \cup \{from\_node\}$
**end if**
$page.permissions = is\_write$ ? readwrite : read
$page.lock = 0$
process $page.wait\_queue$

**invalidate**($requesting\_node$, $page$): server for **invalidate** messages

**if** no forward progress since $page$ received **then**
  enqueue message on $page.wait\_queue$
  schedule callback to process $page.wait\_queue$
**else**
  $page.permissions =$ nil
  send **invalidate_reply**($this\_node$, $page$) to $requesting\_node$
**end if**

**invalidate_reply**($from\_node$, $page$): server for **invalidate_reply** messages

$page.copyset = page.copyset \setminus \{from\_node\}$
**if** $page.copyset = \emptyset$ **then**
  $page.permissions =$ readwrite
  $page.lock = 0$
  process $page.wait\_queue$
**end if**

# Chapter 8

# Performance challenges

In utilising multiprocessor systems, the goal is usually either to minimise the time taken to execute a task or to maximise the throughput of tasks. As an example of the earlier case, assume that an easily-divisible computational task executes in $\tau$ seconds of continuous runtime on a single processor, which might correspond to a certain number of processor operations. Ideally, one might hope to be able to divide the task equally between $N$ processors to achieve a time of $t_{ideal} = \frac{\tau}{N}$, which over the $N$ processors similarly sums to $\tau$ total processor time. In practice one generally finds that the actual execution time is somewhat greater, $t > t_{ideal}$. In this longer time $t$, there is now potentially $Nt$ of processor time available. This implies that some of this processor time ($\tau_{overhead} = Nt - \tau$) is used unproductively. The challenge is to identify and minimise this lost time, so that it can instead be used for productive computation.

In a distributed shared memory system such as that described so far, the lost time is primarily attributable to waiting for responses in the DSM protocol. Whenever a fetch or invalidation message is sent, execution on the local processor must stall until the response is received. As will be seen in Chapter 12, the round-trip time in a typical Gigabit Ethernet network is around 17 $\mu$s, even after extensive tuning of the network driver. On 900MHz Itanium systems this corresponds to over 15,000 wasted processor cycles for each such communication event. When a 4 KiB page of data must be fetched, this increases to around 62 $\mu$s, or 56,000 cycles.

There are three main approaches that can be applied to better utilise compu-

tational time: using lower-latency networking technologies that reduce the time for each communication event, masking the latency of communication events via multithreading, and applying more sophisticated DSM techniques that can reduce the number of events that stall execution.

Indeed there are a number of low-latency high-bandwidth clustering technologies available, such as Myrinet [74], InfiniBand [45] and SCI [23], all of which claim round-trip latencies of under 5 $\mu$s as well as very high bandwidth. Interestingly, the latest generation of Myrinet is based on the 10 Gigabit Ethernet standard, which suggests that similar performance may be achieved by other future 10 Gigabit Ethernet chipsets. However, all of these technologies are currently specialised and expensive. One of the central philosophies of vNUMA is to leverage commodity hardware, and to investigate whether novel approaches can allow commodity hardware to supplant expensive specialised hardware for certain applications. For this reason, this thesis primarily considers Gigabit Ethernet. There is no doubt, however, that a faster hardware interconnect would improve the raw performance of vNUMA. Section 12.2.6 presents a latency sensitivity analysis which can be used to infer possible performance improvements from faster networks.

When long latencies are unavoidable, multithreading is a common technique to reduce the amount of computational time that is wasted: by switching to a different thread when a stall occurs in one thread, some of the stall time can potentially be used usefully. This technique is used by some multithreaded DSM systems, such as Distributed Filaments [32], Brazos [85] and DSM-Threads [73]. It is used by a number of modern CPUs, including the Itanium 2 "Montecito" processor [67]. Indeed the same technique is also used by operating systems to hide I/O latency. At the vNUMA hypervisor level, these threads would correspond to extra virtual processors, beyond the number of physical processors. There are, however, drawbacks to such an approach. For instance, tasks must be distributed between even more processors, introducing extra overheads. Resources (such as locks) held by processors that are not currently running could delay running processors. For some workloads, the virtual threads co-habiting a node could frequently stall simultaneously, negating any benefits. A multithreaded approach seems a promising avenue for future research, but it is no substitute for reducing

the frequency of stalls.

This thesis specifically focuses on reducing the frequency of stalls by using better DSM protocols that leverage relaxed consistency models. The previous section already presented a number of simple protocol optimisations beyond the basic Li-Hudak protocol, however there are fundamental limitations due to the atomic consistency model. If one permits a more relaxed consistency model, further improvements are possible. This will be explored in the following sections.

## 8.1 Addressing sparse accesses

Minimising the number of communication events in a distributed shared memory system depends critically on caching. Many commonly used data structures, such as linked lists and trees, tend to have poor spatial locality, and may result in a processor accessing many pages. If locally cached copies of these pages can be accessed, then overheads are small, but if each of the pages regularly requires a remote fetch, performance will suffer greatly.

In the absence of writes, pages eventually become read-shared, allowing each processor to access the cached copy of those pages without any communication. This is clearly desirable. Now consider that some processor occasionally writes a value to a certain page that is otherwise read-shared. In the Li-Hudak protocol, first the writer must stall while all of the read copies are invalidated, then all of the active readers eventually stall and re-fetch the entire page data. Clearly it would be more efficient, for such sparse updates, to simply propagate the write to any readers.

Related to this sparse write problem is the problem of *false sharing*. This occurs when two unrelated data items happen to co-exist on the same page, and different processors inadvertantly write to these unrelated data items at the same time (or one writes and another reads). The result is contention and unnecessary communication, being a side-effect of the page granularity rather than a necessary data integrity measure. Such false sharing can be addressed with a multiple-writer protocol which permits multiple simultaneous writers (and readers) to a given page, or at least a multiple-reader single-writer protocol which permits a writer

to co-exist with readers. However, both of these require some relaxation of the consistency model.

The PRAM consistency model — the most relaxed uniform consistency model that was considered in Section 6.2 — provides some insight as to what an ideal solution may look like. PRAM consistency was motivated by a scenario where reads can always execute from a locally cached copy of a page, while writes are lazily distributed to other nodes. In such a system, neither reads nor writes must stall. The only restriction imposed by PRAM consistency is that writes must become visible in the same order they were issued by the writing processor. If one imagines a simple implementation in which each write is sent within a message and applied upon receipt, this corresponds to the network providing ordered delivery.

In fact, it turns out that such a design can quite easily provide stronger consistency than PRAM consistency. For example, if the underlying network layer provides causally ordered delivery, then the resulting consistency model is causal consistency. Indeed, many network topologies that provide ordered delivery also naturally provide causally ordered delivery; this will be illustrated for Ethernet in Section 9.1.

Further, if the message delivery system provides total order broadcast, then one can even achieve sequential consistency. This was first proven by Attiya and Welch [5]. For some Ethernet topologies one can indeed obtain a limited form of total order broadcast without protocol overhead (again the details are in Section 9.1). However, in order to attain sequential consistency, the Attiya-Welch protocol requires that a writer must immediately broadcast each write and stall until it receives its own broadcast, thus guaranteeing that it observes the write in the same order as other nodes. (Such an execution satisfies the WW constraint for sequential consistency described in Section 6.2.) This would be prohibitive on Ethernet, both because it implies that each write must occupy a separate Ethernet frame, and also because Ethernet broadcasts are not normally received by the sender.

If the stall that is required by the Attiya-Welch protocol is omitted, the ramification is that a processor can observe its own writes before they become visible to other processors (and thus observe a different relative order). However,

this is also the case for real processor architectures that feature write buffers: reads on the local processor can receive values *bypassed* from write buffers before they have become globally visible. Thus, even though a system based on non-stalling broadcast of updates does not strictly obey sequential consistency, it can reproduce the memory consistency behaviour of a processor with write buffers, as embodied by the TSO consistency model. This is the principle on which the vNUMA write-update protocol is based.

The following sections will describe the write-update protocol in more detail. In particular, Sections 8.2 and 8.3 describe how writes are intercepted and transmitted to other nodes. Since this interception incurs overheads, it is not appropriate for all pages; Section 8.4 describes an simple adaptive scheme which decides between write-update and write-invalidate modes on a per-page basis.

While the consistency model that results is generally indistinguishable from the presence of local write buffers, there are two important differences. Write buffers have the property that they are always flushed *eventually*, and thus the value of a variable always converges to a globally coherent value. This property is not guaranteed by the write-broadcast protocol as it stands; the problem and a solution that can restore coherence is presented in Section 8.6. The Itanium architecture also provides a memory fence instruction that can, amongst other effects, restrict bypassing from write buffers. The implications of this are discussed in Section 8.9.

Additionally, the handling of atomic read-modify-write operations to shared memory pages must be considered. A naïve approach is to fall back to the core invalidation-based protocol for such accesses, but this can degrade performance, considering that these atomic operations are often used in concert with widely shared data structures. A better protocol is described in Section 8.7.

Finally, the evaluation in Part III of this thesis provides benchmark results that demonstrate the effectiveness of these protocols.

## 8.2 Write detection

Multiple-writer protocols such as the vNUMA write-update protocol allows writes to a page to simultaneously complete on multiple nodes. Eventually, all of these

writes must be merged together to produce a consistent version of the page. Thus it is not sufficient to simply transfer entire page data in update messages, since it would be impossible to know how to safely merge two versions of a page created by different writers; aside from which, transferring entire pages is wasteful when only small parts of a page are written at any time. Instead, it is necessary to detect the individual writes that have been made to pages during a given interval, and then apply these writes in a safe order on each node that requires them.

There are three basic approaches for detecting the writes made to a memory page:

- *Software write detection*: The compiler and runtime system are modified to attach instrumentation to write instructions. This scheme was used in the Midway DSM system [92].

- *Page diffing*: Pages are write-protected. Upon a write fault, a 'twin' copy of the page is created and the application is freely allowed to write to the page. When updates need to be propagated, the new page is compared with the twin to determine the changes that have been made. This scheme was used in Munin [9] and many later systems.

- *Write trapping*: Pages are write-protected. Upon a write fault, the faulting write instruction is disassembled and the data to write is determined. The page is left write-protected and the program continued at the next instruction. This scheme was also considered by the Munin authors [10], but was presumed to result in excessively high runtime overheads.

The software write detection approach requires applications to be compiled with a special compiler. This is not a good choice for vNUMA, since one of the goals of vNUMA is to allow transparent distribution of legacy applications and operating systems, which may be written in an assortment of languages.

Out of the two remaining approaches, based on hardware protection, the page diffing approach is the one that is normally chosen for software DSM systems. Its main advantage is that it avoids faulting to software on each write. Nonetheless it also has significant overheads since a copy of each page needs to be made and then compared, taking many cycles of processor time and polluting the cache.

(For sparse writes to many pages, this per-page overhead can be significant; for many writes to a single page, it may be more desirable to simply invalidate the page and propagate the new data as a whole.)

Regardless of these trade-offs, the page diffing approach cannot be used for vNUMA, for the following reasons. Firstly, by the time that the diffing is performed, information has been lost about the size of the writes, which has implications for the outcome of conflicting writes. For example, consider the program execution in Figure 8.1, in which P2 writes -1, P3 writes 1, and then P1 issues a read. The Itanium architecture dictates that the outcome will be one of 0, -1 or 1 (depending on which of the writes have been seen at P1). However, the diff generated at P3 may contain as little as one byte, since in binary representation only one byte of the value has changed. After both diffs are applied, the value at P1 may be 0xffffff01, which is neither of the expected outcomes. Diffing at a 32-bit granularity would solve this problem for 32-bit values, but there would still be problems with smaller and larger types. Systems that employ diffing, such as TreadMarks [50], rely on the programmer to avoid issuing conflicting writes within an interval, and to take care when using smaller types than the diff granularity. However, the Itanium architecture does not have such a requirement; in fact the code example is completely legal if the programmer does not require a guarantee as to which change is applied first. This would present problems for legacy code on vNUMA.

Secondly, the standard diffing approach involves making the page freely writable, hence avoiding further write faults. However, if a page is both readable and writeable, then atomic read-modify-write instructions such as compare-and-exchange will freely execute. Allowing such instructions to execute on a multiple-writer page destroys their very semantics; they are no longer atomic with respect to the shared memory. These instructions must be intercepted and performed in an exclusive fashion. Unfortunately, the Itanium architecture does not provide any mechanism that would allow these specific atomic instructions to be intercepted, short of either making the pages uncacheable or temporarily denying read access to multiple-writer pages, both of which have significant performance implications in themselves. Once again, user-level DSM systems that employ diffing schemes can simply pass this restriction on to the programmer,

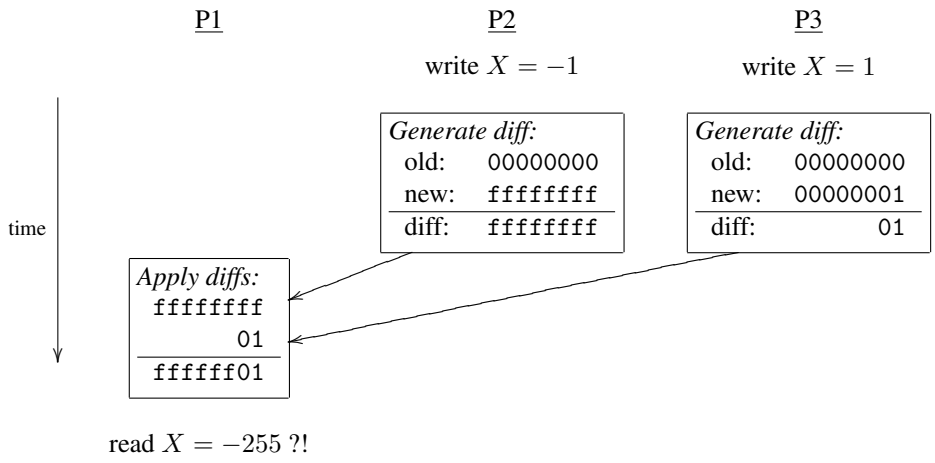



**Figure 8.1:** Problem with write diffing in the presence of conflicting writes. Assume that the initial value of $X$ is 0.

by stating that the programmer must use the synchronisation constructs provided by the DSM system, and not rely on the behaviour of atomic instructions to shared memory. This is not practical for vNUMA.

For these reasons, a write trapping approach has been implemented for vNUMA. Since vNUMA is a thin hypervisor running on bare hardware, the cost of this write trapping can be kept to a minimum. The current C language implementation results in an overhead of around 250 cycles per write, but this is largely due to compiler limitations; in theory under 100 cycles should be achievable. It should be noted that this cost is only incurred for sparse writes to pages in write-update mode; the adaptive protocol in Section 8.4 causes write-invalidate mode to be used if there are many writes to be made to the one page. Thus, the overheads are quite reasonable in practice, as demonstrated by the performance results in Part III.

## 8.3 Write propagation

Once a write has been detected, it is first necessary to apply it to the local copy of the page, if the page is readable on the local node. This ensures that the write

is immediately visible to subsequent local reads, which is a standard programmer assumption and a requirement of all practical consistency models. In relation to the hardware consistency model, this is analogous to the situation in which the write is in local processor write buffers but not yet globally visible.

Program execution then continues, and additional writes may be accumulated. At some point it becomes necessary to 'flush' the write buffers and make the writes visible to other nodes. This may either be because the write buffers are full, or a certain time has elapsed, or for consistency reasons; the question of when and why it is necessary to flush write buffers will be discussed in Section 8.8.

There are then two options for propagating the writes to remote nodes: write-invalidate or write-update. In a write-invalidate scheme, the local node could acquire each page exclusively in order to apply the queued writes; this is similar to the hardware model in which cache lines must be acquired exclusively in order to apply the writes present in write buffers. This is still advantageous over the original write-invalidate protocol which does not buffer writes, since program execution need not be stalled while this occurs. However, it does not adequately address one of the original problems that these improvements were intended to address: in the presence of false sharing and sparse writes, such solutions perform poorly because many invalidations occur.

Instead, a better alternative is to eagerly propagate the writes to other nodes; either via broadcast or to specific nodes. In vNUMA, the write-broadcast technique was chosen.

There are two advantages of broadcasting updates. Firstly, it greatly simplifies the process of maintaining correct write ordering; there is no need to consider nodes which have received some past updates and not others. In particular, it makes it practical to implement consistency models which rely on a total ordering of writes, such as TSO. Secondly, it avoids an explosion in the number of messages when used in conjunction with the migratory DSM protocol used in vNUMA. The issue arises because the copyset of the page is only known by the owner, and the manager must be consulted to determine the owner. Given a set of updates, the sender would first need to split them by manager and send each subset to a manager. Then, each manager splits the updates by owner and sends each subset to each owner. Finally, each owner propagates updates to its copyset.

Naturally some optimisations could be made, however clearly such a protocol is more complicated and uses more messages than a simple broadcast of updates, as well as making ordering guarantees difficult.

It may seem wasteful to broadcast all write updates rather than propagating the updates only to subscribers, and certainly this is one factor that could limit scalability to large numbers of nodes. However, extreme scalability is not a goal of vNUMA. Most importantly, write updates are only used in the case of sparse writes to a contested page, otherwise the page is fetched exclusively (as described in the next section); thus it can be assumed that a number of other nodes have an interest in the page. Furthermore, each write only occupies 16 bytes and many writes are batched into a single message; if each node is interested in *some* of the writes contained in each message, the incremental cost of having some ignored writes in a message is small. Once again, the performance results in Part III prove that this approach is quite successful.

## 8.4 Adaptive hybrid protocol

In vNUMA all memory pages are part of the shared address space. Intercepting individual writes is a good strategy for sparse write patterns that touch small areas of many pages. However, if many writes are to be made locally to a single page, then it is preferable to fetch the page for exclusive access. In that case, the cost of the fetch is amortised over many writes and is less than the cumulative overhead of intercepting each individual write and propagating it to other nodes. This suggests a hybrid scheme in which the write-update scheme is used for sparse writes and a write-invalidate scheme is used for dense writes.

Of course it is not possible to determine *a priori* how many writes are to be made to a page when it is first accessed. Like many control problems in the computing discipline, future behaviour must be inferred from available information about past behaviour. It is necessary to choose an initial algorithm, and then continuously adapt based on the observed write pattern.

In choosing a decision function for adaptation it is necessary to take into account three factors: it should choose write-invalidate mode when many writes are being made by a single node; it should choose write-update mode when there

are sparse write patterns and/or false sharing; and the overhead of tracking page statistics for the algorithm should be minimal.

In particular the scheme that was initially chosen is based on the RWB (Read Write Broadcast) protocol described by Rudolph and Segall [82], from the domain of hardware cache coherency research. Each page starts in write-update mode: thus writes trap and it is trivial to maintain a counter of local writes to the page. This counter is reset whenever a remote write arrives; hence it tracks the 'run length' of local writes uninterrupted by remote writes. When the counter reaches a certain threshold ($T$), it can be assumed that either the local node is making many consecutive updates, or the page is simply used by that node only. In either case the page is acquired exclusively (write-invalidate) and further writes complete without trapping.

The optimal value for the threshold $T$ depends on the relative cost of invalidating the page compared to the cost of intercepting writes individually. However, the real cost of invalidating the page is difficult to determine, because it depends on whether the page will be needed again on another node. The effect of threshold choice is analysed experimentally in Section 12.2.2.

Aside from the case where the threshold is reached, there are also three other types of accesses that can cause a page to immediately fall back to write-invalidate mode.

- atomic read-modify-write instructions,
- certain rare store instructions which are not handled, such as 10-byte floating point stores, and
- references made by the Itanium register stack engine — these are difficult to emulate, and stack pages should in any case be used exclusively.

For the first case, a better solution will be presented in Section 8.7.

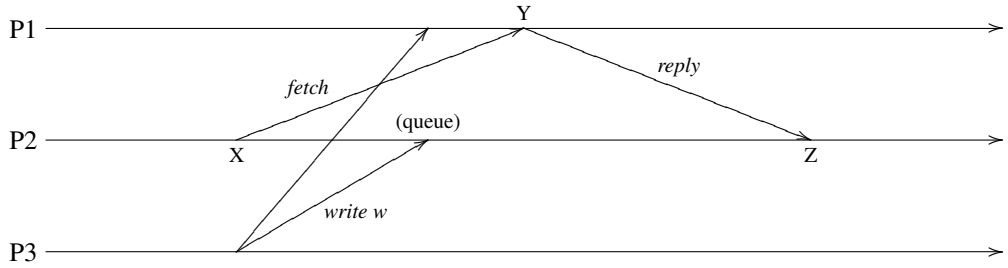
When transitioning to write-invalidate mode, acquiring page ownership from the previous owner is done synchronously — as in the core protocol — which ensures that that multiple nodes cannot simultaneously claim ownership. However, invalidations can then be propagated without waiting for replies. Propagating invalidations is similar to propagating updates, so there is no relaxation of the consistency model.

One point to note is that there is no global consensus on which mode a page is in: each node independently decides whether to intercept writes and send write notices, or whether to attempt to acquire the page exclusively. Further, even if one node owns a page exclusively, other nodes can still write to that page (as long as they do not read from it); the write notices are applied by the exclusive owner on behalf of the other nodes. This feature may be useful for certain scenarios where one node accesses a page frequently and other nodes write occasionally.

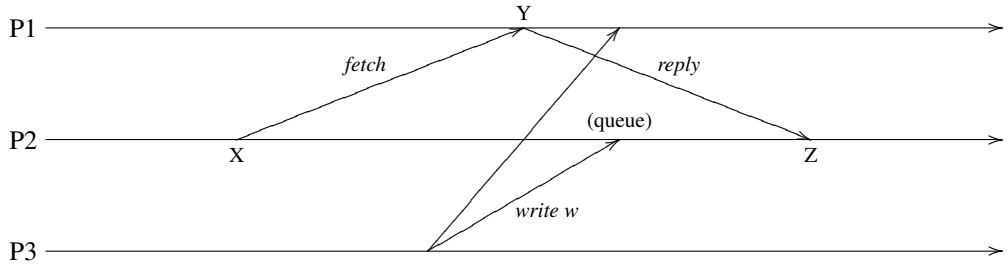
An unfortunate problem with the RWB scheme is that read-write sharing is not detected. A page that is written by one node and read by other nodes will be periodically acquired by the writer each time its write counter reaches the threshold; then the readers re-request the page and the page returns to write-update mode for another cycle. This oscillation is clearly undesirable. A later protocol, the Efficient Distributed Write Protocol (EDWP) [4], solves this problem by tracking both read and write accesses on each node, and preventing a transition to exclusive mode if more than one processor is accessing a page. Tracking read accesses does present complications in a software DSM system, however; it is necessary to remove read permissions and then observe if a fault occurs. Additionally, the EDWP protocol invalidates some read copies even if a consensus to invalidate is not reached globally, which may have detrimental impact on performance. A better decision function, one that is more appropriate for the vNUMA system while detecting read-write sharing, could be the subject of future work.

## 8.5 Interaction with migration

Complications arise in the interaction between the write-update protocol and page migration. When a node is waiting to receive a page, should it discard any updates it receives for that page, or should it queue them and apply them to the new copy of the page that it receives? Figure 8.2 shows that the answer to this is not a simple one. Assume node P2 is fetching a page from node P1; whether node P2 needs to apply certain updates depends on whether node P1 already received those updates before sending off the page. Further, since node P1 was originally the owner, it may have made local changes in the period between applying an update and




(a) Write $w$ arrives at P1 before Y, must be discarded at Z




(b) Write $w$ arrives at P1 after Y, must be applied at Z

**Figure 8.2:** Timelines showing interaction between migration and write updates

sending the page data, so updates must not be re-applied a second time if they have already been applied. A decision algorithm will be presented here.

### Derivation of algorithm

It will be assumed here that the network or network protocol provides causal order delivery (as described in Section 9.1). Consider the events the events $X$, $Y$, and $Z$ as shown in figure 8.2, and any write $w$. According to causally ordered delivery, because $X \rightarrow Y$, $(w \rightarrow_2 X) \Rightarrow (w \rightarrow_1 Y)$ [1]. In plain language, any write that is observed before X at P2 must be observed before Y at P1 (but note that additional

[1]Following notation used in many other sources, $a \rightarrow b$ states that $a$ precedes $b$ globally, and $a \rightarrow_p b$ states that $a$ precedes $b$ locally on processor $p$.

writes may also be observed at P1 by Y). Since this holds true for every write, given a counter at each node that counts observed writes, the value of the counter at Y (at P1) must be at least that at X (at P2), i.e. $C_Y \geq C_X$. The difference $C_Y - C_X$ corresponds to the additional writes that were not yet received by P2 at X, but are applied at P1 while the fetch request is in transit.

Now consider the reply phase. By the same argument, $(w \rightarrow_1 Y) \Rightarrow (w \rightarrow_2 Z)$ and $C_Z \geq C_Y$. The difference $C_Z - C_Y$ corresponds to the set of writes that are received at P2 by Z, but were not applied at P1 before Y.

Clearly all of $\{C_X, C_Y, C_Z\}$ can be known to P2 at Z: $C_X$ and $C_Z$ are local counter values, and it is easy enough to include $C_Y$ in the fetch reply message. Thus, P2 knows how many writes have already been applied to the page ($N_{drop} = C_Y - C_X$) and how many have not ($N_{apply} = C_Z - C_Y$), which must total the number of writes that were queued ($N_{queued} = C_Z - C_X$)[2].

Now that it is known *how many* writes must be applied, what remains is to determine *which* writes to apply. To determine this, it is again necessary to consider the message ordering provided by the network. Given ordered delivery, the writes from each individual processor arrive in order. Thus if one considers each source processor separately — with a separate counter for each — then it is simply a matter of dropping the oldest and applying the newest writes *from each processor*. Such a set of per-processor event counters is similar to a vector clock [64].

The problem with vector clocks is that they are large structures that are unwieldy to send over the wire with each fetch reply. Therefore a simpler solution is desirable when possible. If the network topology provides a form of total order broadcast, then a vector clock becomes unnecessary, as there is a global total order which can be leveraged. The solution then is trivial: the earliest $N_{drop}$ writes should be dropped and the latest $N_{apply}$ writes should be applied.

The vNUMA network does not provide true total order broadcast, but it does provide a form of sender-oblivious total order broadcast in which the sender observes broadcasts earlier than the total order (see Section 9.1). In this case, the

[2] In fact the situation is slightly more complicated because in some cases the fetch request may be routed via another node: the manager. However, this does not invalidate the argument, since $X \rightarrow Y$ is still true.

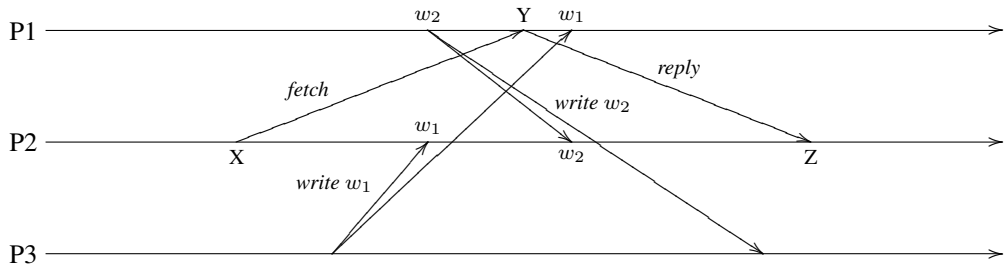



**Figure 8.3:** Timeline showing a possible ordering problem (see text)

trivial total order solution does not hold. Figure 8.3 shows a problematic timeline. Comparing write counters ($C_X = 0$, $C_Y = 1$, $C_Z = 2$) shows that one write should be applied and one should be discarded. The trivial solution would discard the oldest write $w_1$. However, careful examination of the timeline shows that in fact $w_2$ should be discarded — it was already applied at P1 — and $w_1$ should be applied, since it only arrived at P1 after Y.

The problem here is that the sender-oblivious total order broadcast scheme does not guarantee the order of writes originating at one of the observing nodes; thus one needs to separately consider the writes originating at P1 and P2:

- Writes by P1 before Y: all of these writes arrive at P2 before Z (ordered delivery), and they can all be dropped! Hence, in Figure 8.3, it is known that $w_2$ must be discarded, leaving $w_1$ to be applied.
- Writes by P1 after Y: these necessarily arrive after Z (ordered delivery), and are applied normally.
- Writes by P2 before X: these necessarily arrive before Y (causally ordered delivery), and are applied normally.
- Writes by P2 between X and Z: one might assume by ordered delivery that these would arrive after Y, but given a situation in which the fetch request must be routed via another node (a manager), it is possible that a write after

X could arrive before Y. To avoid this situation occurring, sending writes while waiting for page data is disallowed[3].

- Writes by P2 after Z: these necessarily arrive after Y, and are applied normally.

### Final algorithm

Therefore, a final algorithm can be stated for processing the set of queued writes once the page data is received at Z:

1. All writes from the page sender are dropped (assume there are $N_{sender}$ of these) — it is known that these are included in the page data.

2. Out of the remaining writes, the earliest $C_Y - C_X - N_{sender}$ writes are dropped; equivalently the latest $C_Z - C_Y$ writes are applied. ($C_X$, $C_Y$ and $C_Z$ are counts of received writes at X, Y and Z respectively, with $C_Y$ sent in the fetch reply.)

Note that it is not necessary to consider any later writes that are received at P2 after Z, because these writes necessarily arrive at P1 after Y (the contrapositive of $(w \rightarrow_1 Y) \Rightarrow (w \rightarrow_2 Z)$ is that $(Z \rightarrow_2 w) \Rightarrow (Y \rightarrow_1 w)$). Further writes should simply be applied at both processors as normal.

## 8.6 Coherence

The write-update algorithm presented thus far is sufficient to run Linux and all of the tested applications. It does, however, have a theoretical problem: it does not fully guarantee coherence. Simultaneous writes by multiple nodes to the same location may result in the writers disagreeing as to which value is the final value. This is rarely a serious problem in practice, because each node individually observes a consistent state of affairs. Nonetheless, it would be desirable to find a solution.

[3]Since this is overly restrictive for some of the vNUMA optimisations, the present algorithm is actually implemented with per-page counters rather than global counters, and thus only writes to the particular page that is being waited on need be suppressed.

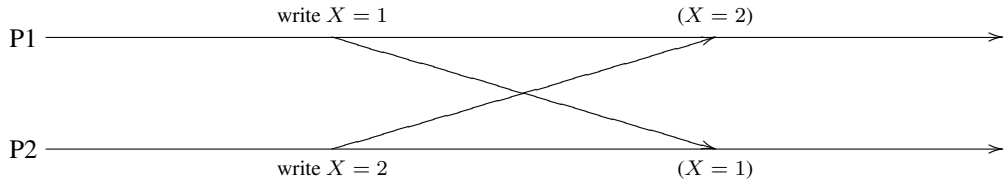



**Figure 8.4:** Coherence problem with write notices

To see how the problem can occur, consider the simple timeline presented in Figure 8.4, in which two processors simultaneously write different values to the same read-shared location. Each node first observes the value that it wrote locally, which is consistent with the behaviour of processor write buffers. However, both nodes then apply the remote writes. In the absence of further writes, P1 now has $X = 2$ and P2 has $X = 1$. Further, it can be seen that this situation appears symmetric to both nodes: even though the network might provide (sender-oblivious) total order broadcast, neither node observes its own transmission and thus neither node can authoritatively determine a total order. It is up to the protocol to provide a solution that provides for a true total ordering.

Indeed the total order broadcast problem is a common problem in distributed systems, so existing work provides some insight into how this could be achieved. There are two general approaches presented in the literature, as comprehensively summarised by Defago et al [21]: protocols that use a central sequencer, and deterministic merging protocols that reorder messages into a deterministic order at each receiver.

A central sequencer does partially resolve the problem of Figure 8.4. Assume that P1 and P2 must send their updates via a sequencer S. The result is shown in Figure 8.5. Ultimately, both processors agree on the final value $X = 1$, so this does solve the problem of shared memory divergence. However, note that P1 observes $X = 1$ followed by $X = 2$ followed by $X = 1$, even though $X = 1$ only appears once in the program. (It is not possible to delay the visibility of the local write and continue the program.) Strictly speaking this violates coherence and

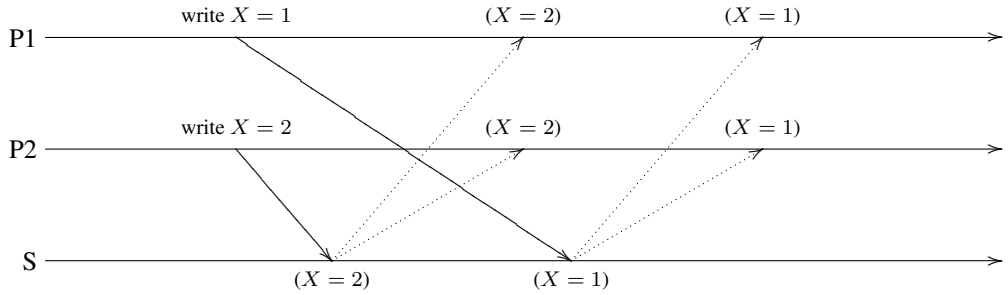



**Figure 8.5:** A central sequencer prevents divergence but does not provide true coherence

is disallowed by the Itanium architecture. In addition, a central sequencer could clearly become a bottleneck.

Deterministic merging allows for a better solution. Consider assigning every write a globally unique sequence number $S(w)$, and including that sequence number when the write notice is transmitted on the network. Every node — including the sender — can then reconstruct a unique ordering $w_1, w_2, ..w_n$ which consists of the writes sorted by sequence number, i.e. $S(w_1) < S(w_2) < ... < S(w_n)$.

In order to guarantee that every write has a globally unique sequence number, one first ensures that uniqueness holds locally by using a monotonically increasing counter. Then, one can use the node number as a secondary differentiator (for instance, by placing it in the low-order bits of the sequence number). Where there is a collision between counter values on different nodes, the node number resolves the conflict. In order to preserve causality — the property that dependant writes follow each other in the global order, in other words have a higher sequence number — the counter used is a logical clock, and is synchronised on communication. This is similar to the technique used in the mutual exclusion algorithm presented by Lamport [55].

Of course, in order to calculate the total order $\{w_1, w_2, .., w_n\}$, one must first wait to receive write messages from all nodes. In vNUMA, write notices are only sent when necessary, so there could be long periods when a particular node is

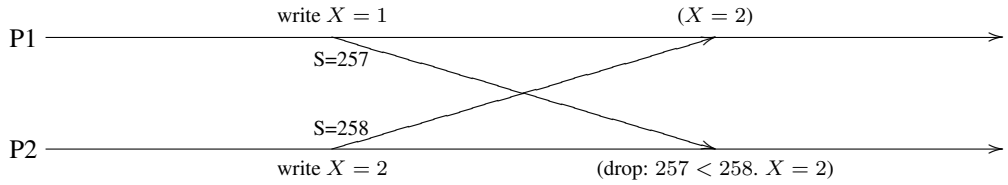



**Figure 8.6:** Deterministic merging to ensure coherence. Both nodes ultimately agree on $X = 2$, the write with the higher sequence number. A method for choosing unique sequence numbers is described in the text.

quiet. In order to guarantee liveness and reduce latency, it would be necessary to regularly send empty messages from each node.

However, coherence only requires total ordering on a per-location basis. Consider the case where $\{w_1, w_2, .., w_n\}$ are a set of writes to the same location. From the point of view of program semantics, it is not essential to guarantee that all of $\{w_1..w_n\}$ are observed, as long as the observed subset follows the correct ordering and culminates in the proper final value. In other words, observing $\{w_2, w_1, w_n\}$ is not allowed since $S(w_2) \not< S(w_1)$, but observing $\{w_1, w_n\}$ or even just $\{w_n\}$ is allowable. In practical terms one could simply assume that that processor was not fast enough to observe the intervening values.

This suggests a technique which will be referred to as *incremental* deterministic merging. Each incoming write notice is applied immediately, but it is only applied to a certain location if its sequence number is greater than the last write to that location. Since every node receives all write notices, the value of that location always ultimately converges on the write with the maximum sequence number ($w_n$), with any intermediate values respecting the required ordering. Figure 8.6 shows how this resolves the original problem.

### Practical implementation

In the preceding discussion the definition of a location was deliberately vague. In a real computer architecture, it is possible to issue overlapping writes with

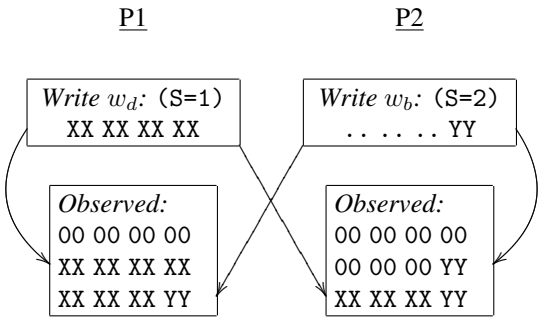



**Figure 8.7:** Combining writes of different sizes. On P2, write $w_d$ appears to modify 3 bytes.

different sizes. For example, consider a 4-byte write $w_d$ with $S(w_d) = 1$, and a byte-sized write $w_b$ with $S(w_b) = 2$ (thus $w_d$ precedes $w_b$). If the newer byte-sized write happens to be applied first at some node, then when the older 4-byte write is received, it must only appear to modify the top 3 bytes, as illustrated in Figure 8.7. This set of observed values is consistent with the Itanium memory consistency model [46].

This illustrates that conflict resolution must be applied at the byte level and not the word level. Conceptually the solution is to store a sequence number for each byte. As soon as $w_b$ is applied, the sequence number of that lowest byte becomes 2, and thus $w_d$ with sequence number of 1 cannot affect the value of the lowest byte.

In practice, of course, storing a sequence number for every byte of every page is prohibitive; for 16-bit sequence numbers it would reduce the amount of usable memory to a third. Fortunately, it is only necessary to store sequence number information for a short time. Once updates are received from all nodes with at least a certain sequence number, all information related to lower sequence numbers can be discarded, since any new incoming writes will have a higher sequence number.

Since the conflict resolution algorithm is executed for every write update, the data structure used to track sequence number information must be carefully designed. In particular, since the majority of updates received will not conflict, the tracking overhead must be minimised. It must also be easy to clean up data about old sequence numbers that are no longer valid.

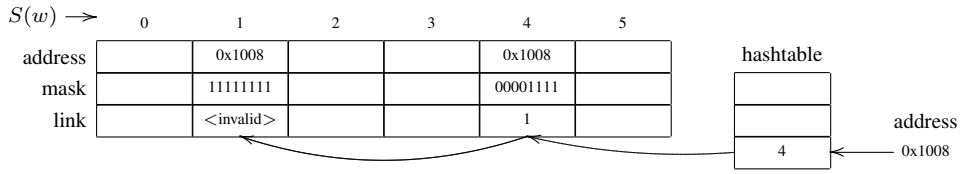



**Figure 8.8:** Data structure used for coherence algorithm

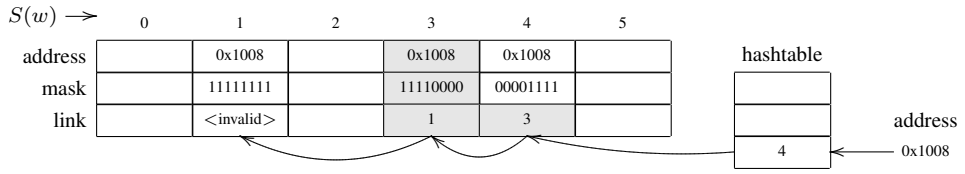



**Figure 8.9:** Adding a new write. The incoming write has sequence number 3, address 0x1008 and mask 11111111 (entire 8 bytes). The hash chain is traversed as far back as sequence number 4; since that logically newer write wrote 00001111 (the lower four bytes), the mask is constrained to 11110000 (the top four bytes). The appropriate slot for the new write is then updated and linked in place.

A number of formats were investigated for this data structure. Early implementations used a pool allocator to allocate tracking structures on demand, where each tracking structure contained dense sequence number information for a small fixed-size area of memory. These tracking structures were then added to two lists: a per-page list, used for lookup, and a clean-up list. The per-page list was doubly linked so that the clean-up process could later unlink the tracking structure from the per-page list.

However, such implementations suffered from large overheads, because of the complexity of the data structures and the cache misses caused by their sparseness. A better implementation was designed, based on a data structure quite similar to that introduced in Section 5.2.2.

In the improved implementation, a fixed size buffer stores information about a certain number of preceding writes (Figure 8.8). Writes are directly inserted into the buffer using the least significant bits of their sequence number as an index; assuming that sequence numbers are allocated in a unique and relatively dense

fashion, this mapping is quite efficient. For fast lookup, writes are then indexed using a hash function of their target address; writes with the same hash value are linked together in a chain. This chain is always kept in reverse sequence number order.

There is only a single operation on this data structure: adding a new write (Figure 8.9). The arguments are the sequence number of the new write, a word-aligned address[4], and a mask of bytes to write within that word. The hash chain for that address is looked up and traversed as far back as the given sequence number; this path encounters all logically newer writes, which are used to further constrain the mask (in cases where the address matches). Once a link field with an older sequence number is reached, traversal stops and the new write is inserted into the chain. The constrained mask is returned and used to determine the bytes in memory that should be written.

Since a chain is never traversed past the sequence number of a newly received write, the chains need never be garbage-collected. It is sufficient to ensure that the buffer is sized large enough so that it covers the window of valid sequence numbers that can be received at any one time. Since each node tracks the last sequence number received from each other node, a violation of this rule can be detected and a stall induced if necessary; however such stalls are clearly undesirable.

This algorithm is simple and has relatively low overheads. The cost will be quantified in Chapter 12.

## 8.7 Atomic operations

In practice, locks and similar synchronisation primitives constitute a major cost in many vNUMA workloads. This becomes particularly problematic within the Linux kernel, since the same kernel necessarily runs on all CPUs. Large locking overheads in the kernel can severely hamper scalability, even for workloads that are not themselves limited by synchronisation performance.

[4] *Word* is used in this context is to refer to a machine word of 64 bits. It is assumed that writes never cross a machine word boundary, or multiple lookups in the data structure would be necessary.

On an SMP system, locks are normally implemented by using atomic read-and-write operations on shared memory; the write phase marks the lock as locked and a 'simultaneous' read of the previous value is used to verify that no other processors have already acquired the lock (the exact definition of atomicity will be discussed shortly). The Itanium processor architecture provides three operations that can be used for this purpose: the *exchange* (`xchg`) instruction atomically writes a memory location and returns the earlier value; *fetch-and-add* (`fetchadd`) also returns the previous value but adds the argument instead of storing destructively; finally *compare-and-exchange* (`cmpxchg`) is an enhancement of *exchange* that only performs the write if the value read matches a given value[5].

In the write-update protocol described thus far, any of these operations result in a fallback to write-invalidate mode: exclusive page ownership is acquired and all other read copies are invalidated. While this guarantees correct execution of the atomic instruction, it is not optimal. For example, when a lock is acquired on multiple nodes, the entire page contents must be transferred each time. This section describes an extension to the write-update protocol (dubbed write-update+) that allows the update strategy to be used even for atomic operations, eliminating the need to continually invalidate pages containing locks and other atomically accessed variables.

It is firstly necessary to define atomicity more precisely. Recall from Section 6.2 that coherence requires that accesses to a single location must be consistent with the existence of a single total order of operations. Atomicity of a read-and-write instruction to coherent memory means that the read and write operations must be adjacent in that total order; any execution that implies a total order where those operations are not adjacent is disallowed.

To give a practical example, if two processors attempt to acquire the same lock simultaneously, at least one must observe it locked. Consider a lock implemented with exchange operations; each exchange operation $RW$ consists of a read operation $R$ and a write operation $W$. Processor P1 issues $RW_1$ (consisting of $R_1$ and $W_1$) and processor P2 issues $RW_2$ (consisting of $R_2$ and $W_2$). The effect must be consistent with one of two total orders, either $R_1 \rightarrow W_1 \rightarrow R_2 \rightarrow W_2$ (in

[5]Note that both *exchange* and *fetch-and-add* semantics can be implemented in terms of *compare-and-exchange* (sometimes known as *compare-and-swap*), but the reverse is not true.

which case $R_2$ observes the lock as locked due to $W_1$) or $R_2 \rightarrow W_2 \rightarrow R_1 \rightarrow W_1$ (in which case $R_1$ observes the lock as locked due to $W_2$). An execution that allows neither processor to observe it locked — implying that the reads are ahead of the writes in the total order, such as $R_1 \rightarrow R_2 \rightarrow W_2 \rightarrow W_1$ — is disallowed by the atomicity requirement; here $R_1$ and $W_1$ are not atomic.

The fact that one of the two processors must observe the lock's memory location in the locked state (the new state) means that two vNUMA nodes cannot independently execute atomic instructions on the same location. At most, one node can be granted permission to execute atomic instructions on a location at any time; the next node must fetch changes from that node before being granted permission to proceed. This implies that a token-passing protocol is necessary.

Ordinary reads on other nodes, however, can always execute from a cached copy. A read on some processor $p$ may validly observe either the value before an atomic instruction or the value after an atomic instruction, supported by the execution orders $R_p \rightarrow R_1 \rightarrow W_1$, and $R_1 \rightarrow W_1 \rightarrow R_p$ respectively — these execution orders simply represent the threads differently interleaved, with atomicity of $RW_1$ preserved. This means that it is *not* absolutely necessary for the DSM implementation to invalidate read copies when emulating an atomic instruction. Assuming that the current node has the token to perform an atomic operation on the target address, it is safe to perform the operation locally and propagate the write part of the operation in the normal write-update stream. This is the critical observation behind the write-update+ protocol.

However, there are some complications. Unlike reads, writes on other nodes need special care. Assume, as usual, that the initial value of a memory location is zero. Consider if a *fetch-and-add* operation $RW_1$ atomically increments the memory location on one processor, and there is a write of the value 5 (denote this as $W_p(5)$) on another processor. $W_p(5)$ must either become visible before the atomic operation ($W_p(5) \rightarrow R_1(5) \rightarrow W_1(6)$) or after it ($R_1(0) \rightarrow W_1(1) \rightarrow W_1(5)$), never interleaved with it. That is, the valid final values are 6 or 5; the value 1 (that might result from $R_1(0) \rightarrow W_1(5) \rightarrow W_1(1)$) violates the atomicity of $RW_1$. Additionally, it can be seen that observing the intermediate value 1 is only valid in the execution where the final value is 5, and not in the execution where the final value is 6.

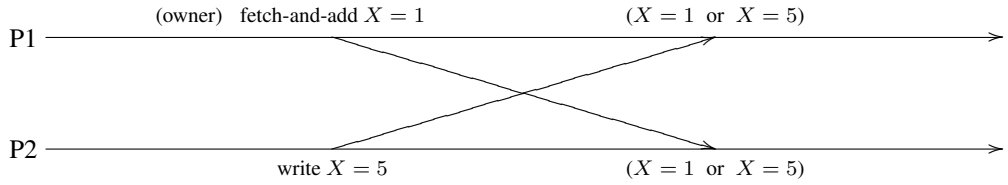



**Figure 8.10:** Problem arising from simultaneous writes and atomic operations. P1 is the owner of $X$ and therefore has the token to execute atomic operations. Without the coherence algorithm, P1 finishes with $X = 5$ and P2 finishes with $X = 1$. With the coherence algorithm, both nodes agree on either $X = 1$ or $X = 5$, depending on the sequence numbers of the two writes. However, $X = 1$ is an invalid result.

Given the vNUMA write-update protocol, a final value of 1 can easily result, as shown in Figure 8.10. This may, at first glance, seem like a problem that should be solved by the write coherence algorithm. However, the write coherence algorithm only guarantees that the value converges to either 1 or 5; it cannot know that 1 is an invalid final value.

It may be possible to resolve this problem with a more complex deterministic merging algorithm, such as by ensuring that writes resulting from atomic operations are always applied earlier in the case of an ordering ambiguity. However, this would require delaying the application of received writes until it is definitively known whether a conflicting atomic operation has occurred; the specific goal of the incremental deterministic merging algorithm described in Section 8.6 is to avoid delaying writes in this way.

To avoid such complications arising from simultaneous writes, the write-update+ protocol enforces a single writer for pages targeted by atomic operations. Thus, at any point, a page can be in one of three modes: write-invalidate, write-update (multiple-writer), or write-update+ (single-writer). The transition from write-update to write-update+ mode occurs when the first atomic operation to the page is intercepted; nodes are synchronously notified that they can no longer generate write updates to the page without acquiring ownership. The benefit of the write-update+ mode, however, is that the owner is now allowed to send write

updates for atomic operations as well as ordinary writes, which will be seen to produce significant performance improvements.

## 8.8 Write batching

For simplicity, the protocol thus far has been described in terms of writes, and writes were generally considered to correspond directly to messages. However, a write notice is small and each Ethernet message has considerable overhead on the wire (headers and inter-packet gap) as well as processing overhead at the receiver. Thus it is clearly beneficial to slightly delay the sending of writes and 'batch' as many as possible into a single Ethernet message.

However, since one node may be waiting on a write that is contained in another processor's write buffers, it is necessary to ensure that write buffers are flushed eventually, and in fact in this case it might be preferable to flush the buffers more eagerly.

Additionally, write batching has implications for the consistency model. If a remote node requests a particular page, any writes that have been made to the page locally will be made visible to that node. If there are other writes which are still being delayed in the write buffers, the writes to the requested page will overtake the buffered writes in visibility order, even though they may have been later in program order. This may violate the consistency model, if the consistency model specifies that writes must be observed in order.

There are two ways that this can be avoided, in order to maintain write ordering:

- Service remote reads from a shadow copy of the page, where the shadow copy represents the remotely visible state of the page at any point in time, and is only updated when write notices are flushed. In this case, write visibility is delayed until the time of the flush.
- Flush all pending write notices (or at minimum the writes that precede any writes that will be made visible) before servicing a remote read to an exclusively held page. Assuming that messages arrive correctly ordered,

this guarantees that the previous writes will become visible before any writes in the read response.

Both of these approaches are problematic because vNUMA does not intercept writes to pages which are in write-invalidate mode, and thus some writes do not pass through the write buffers. The first approach would require shadowing all pages including those in write-invalidate mode; the second would require flushes before every read, since it is not possible to determine the complete set of writes that precede the writes that will be made visible.

However, the Itanium architecture does not require this level of strictness in write ordering. As previously described, the Itanium architecture provides a weaker ordering model based on release consistency. Load instructions can optionally have acquire semantics, guaranteeing that they become visible prior to subsequent accesses. Store instructions can optionally have release semantics, guaranteeing that they become visible after all previous accesses. Atomic read-modify-write instructions can optionally have either acquire or release semantics, but not both. Finally, there is a special memory fence instruction that has both acquire and release semantics.

In fact acquire semantics are guaranteed by processor hardware regardless of DSM behaviour. Consider a read with acquire semantics $R_{acq}$ that is followed by another read or write access $A$ (ignoring memory fences, only reads can have acquire annotations, so this describes a general case). To satisfy the consistency model, $R_{acq}$ must become visible before $A$; the easiest way to guarantee this is to ensure that $R_{acq}$ completes locally before $A$ is issued (recall from Section 6.2 that a read becomes visible before it completes, and by definition $A$ cannot become visible before it is issued). This is necessarily the case for two separate reasons. Firstly, the processor hardware itself interprets the acquire annotation, and the Itanium processor implementation of acquire semantics is such that $R_{acq}$ is indeed forced to complete before $A$ is issued to the memory system. Even if this was not the case, this still holds for a system with one processor per node, because locally accesses always appear to follow program order; the actual order of accesses is only determinable if remote accesses intervene. The only mechanism by which remote accesses occur is via a network interrupt, and interrupts always serialise

execution.

Release semantics require special care, however. This time, consider an access $A$ that is followed by a write with release semantics $W_{rel}$ (only writes can have release annotations). To satisfy the consistency model, $A$ must become visible before $W_{rel}$. The same argument as above can be applied; in particular, the Itanium processor interprets the release annotation and guarantees that $A$ completes before $W_{rel}$ is issued. However, there is a complication: in the case that $A$ is a write, local completion does not imply remote visibility — in the present DSM system, writes may be queued before being propagated to remote nodes. In that case it is up to the DSM system to guarantee that $A$ is observed before $W_{rel}$.

If $A$ and $W_{rel}$ are both writes to write-update pages, then this is trivial: one simply needs to ensure that order is preserved when propagating the write notices. If $A$ is to an exclusively held page, then that write becomes visible immediately (recall that a write-invalidate protocol is atomically consistent), and so there is no question that $W_{rel}$ becomes visible later. On the other hand, if $A$ is a queued write but $W_{rel}$ is to an exclusively held page, then $W_{rel}$ potentially becomes visible first; it is up to the protocol to ensure that $W_{rel}$ is never actually observed before $A$. $A$ can only be observed after a remote node requests read access to that page, so it is sufficient to ensure that $W_{rel}$ is propagated before a read response for $A$.

It only remains to determine how to detect $W_{rel}$ to an exclusively held page, since it does not fault (and indeed cannot be made to fault without making all ordinary writes to that page also fault). To do this, the Itanium performance monitoring unit is used; there is a performance monitoring counter which can be configured to count releases. When a read request arrives for an exclusive page, the counter is checked to determine whether a release occurred on the last interval. If so, the write buffers are flushed before sending the read response. This is a refinement of the second mechanism listed above, reducing the frequency of write queue flushes compared to performing a flush on every read.

Finally, the write queue is eagerly flushed at the time that a write is intercepted, if a release has been seen (either on that instruction or in the previous interval) and if the network card transmit queue is empty. This is an optimisation to expedite transmission of writes, since a release is usually used in the context of data that is intended to be observed by another processor. If the transmit queue is not

empty, then the flush is scheduled to occur after a delay; this rate-limits the update packets and allows additional writes to accrue while previous update packets are being transmitted.

## 8.9 Memory fences

As well as acquire and release annotations — which provide "half fence" semantics, in that they are each permeable in one direction — the Itanium architecture also provides a full fence instruction (memory fence, `mf`) that enforces the strictest possible ordering. Given any access $A$ and $B$ with an interposed `mf` instruction, $A$ must become visible before $B$.

`mf` is counted by the performance monitoring unit as a release (as well as an acquire), so if $A$ and $B$ are both writes then the release detection mechanism described in the previous section guarantees that $A$ becomes visible before $B$. If $A$ is a read, the same argument used in that section can be applied again here: $A$ is always forced to complete locally before $B$ is issued, so $A$ also becomes remotely visible before $B$.

The one case that is problematic is when $A$ is a buffered write and $B$ is a read. Strictly, a memory fence should prevent $B$ from returning a locally cached value, forcing $A$ to become visible everywhere before $B$ completes. An example execution is shown in Figure 8.11. Without the memory fence, the reads on both processors may obtain locally cached values of $X$ and $Y$ (0). The memory fences force the preceding writes to become visible before the reads are issued. If $X$ is the first read performed, then this implies $Y = 1$ is visible by that time, and so the subsequent read of $Y$ must return 1. If $Y$ is the first read performed, then this implies $X = 1$ is visible by that time, so the subsequent read of $X$ must return 1. Thus, the case where both reads return 0 is not allowed.

However, this is difficult to enforce in a protocol such as that in vNUMA. If one considers the situation that both nodes have pages $X$ and $Y$ read-shared and in write-update mode, the writes will result in the local page versions being updated and remote writes being queued (and possibly sent on the network), but neither the `mf` instruction nor the subsequent reads trap to vNUMA so that a stall can be

| P1 | P2 |
|---|---|
| $Y = 1$ | $X = 1$ |
| **mf** | **mf** |
| read $X$ | read $Y$ |

**Figure 8.11:** Example execution illustrating one effect of a memory fence. Assume that the initial values of $X$ and $Y$ are 0. The case where both reads return 0 is not allowed.

enforced[6]. Thus, it is very likely that the reads will return the previous locally cached values, in violation of strict memory fence semantics. The correct values only become visible once the remote writes arrive.

Even if the `mf` instruction could be intercepted, the challenge with such memory fences is that it is not possible to determine, at the time of the memory fence or read, which nodes to communicate with to acquire the relevant updates. P1 does not know that P2 has updates pending to $X$, and that there is a fence on P2 that would force ordering. In order to implement strict memory fence semantics it is therefore necessary either to synchronously communicate with all other nodes and acquire updates, or alternatively to use a global lock that only allows one node to execute a memory fence at any time, preventing the race condition in an execution such as Figure 8.11. Both of these solutions can introduce significant overheads in a large system: in the first case, many messages are required; in the latter case, the single lock becomes a bottleneck.

Fortunately, such strict memory fence semantics are actually rarely needed. Often, full memory fences are used by programmers as a conservative measure, but the code would function correctly with a weaker fence — such as an acquire, release, write fence, read fence, or I/O fence — all of which can be implemented with minor overhead on vNUMA. In cases where a full fence is needed, the same effect can be achieved using atomic operations. Implementing a fence using atomic operations is essentially equivalent to a lock-based implementation of `mf`, but it is far more scalable because the scope is restricted using the address targeted by the atomic operation; it does not require processors executing unrelated fences

[6]It is possible to arrange for a performance monitoring interrupt on the `mf` instruction, but these interrupts are not synchronous with the instruction stream. By the time the interrupt is delivered it is often too late to rectify the bad value returned by the following read, short of rolling back the entire virtual machine to a safe point.

| P1 | P2 |
|---|---|
| try to lock (fail) | |
| *add self to wake-up queue* | *unlock lock* |
| **mf** | **mf** |
| *re-check lock (locked)* | *check wake-up queue* |
| sleep | wake up waiters |

**Figure 8.12:** Use of memory fences in the Linux `wait_on_bit_lock` implementation. Italics show the correspondence with Figure 8.11.

to communicate.

Despite the assortment of synchronisation algorithms implemented in Linux, only one case was encountered in testing which required a full fence — the implementation of the `wait_on_bit_lock` function — and this was resolved via a simple modification. Pseudocode which demonstrates the issue is shown in Figure 8.12. In some memory management data structures, a single bit within a word is used as a lock for the rest of the data structure. In the case that the lock bit is found to be already set, a thread adds itself to a wake-up queue, re-checks the lock bit, and (assuming that it is still set) goes to sleep. Memory fences are used to ensure that either P1 observes the unlock, or P2 observes the write to the wake-up queue (or both, but not neither — if neither occurs then P1 will never be woken up). This is logically identical to the previous example, where the lock and the wake-up queue correspond to $X$ and $Y$.

Notably, the lock attempt and the unlock are already performed using atomic operations, although the position of these operations with respect to the queue operations cannot prevent the undesired outcome. However, if the re-check of the lock is also performed using an atomic operation — it is in any case desirable to lock it if it is found to be unlocked — then atomicity guarantees that this instruction is definitely ordered with respect to the unlock. If the re-check occurs after the unlock, then it observes the unlock, and P1 does not go to sleep. If the re-check occurs before the unlock, then at unlock time P2 will acquire updates from P1, and P2 will necessarily observe the wait queue update (vNUMA *does* enforce the write-write ordering semantics of the `mf`). This resolves the problem.

Clearly this solution violates one of the goals of vNUMA, which is trans-

parency. Nonetheless, the problem arises sufficiently rarely that it would be unfortunate to introduce large overheads intercepting and synchronising memory fences when the majority of uses of memory fences do not rely on strict write-read ordering. Better solutions to this problem should be a subject for future work.

## 8.10 Pseudocode

In this section, the pseudocode in Section 7.6 is extended to reflect the enhancements described in this chapter. As before, a page fault causes the **fault** handler to be invoked. The fault handler now additionally takes an $is\_rse$ flag which indicates whether the access was caused by the Itanium register stack engine (since these accesses cannot be emulated easily). If the faulting operation is a write access and the conditions hold for the write-update algorithm to be used, **fault** calls **emulate_store_operation** or **emulate_atomic_operation**. These procedures acquire ownership of the page if necessary, apply the operation to the local copy of the page and queue a **remote_store** message to be sent to all other nodes. The ultimate receiver of these messages executes the code shown in **handle_remote_store**. Whenever page data or ownership is required, **acquire_page** is called, which is essentially the same as the original write-invalidate protocol — **fetch** or **invalidate** messages are sent as appropriate.

In addition to the per-page metadata previously described ($page.owner$, $page.copyset$, $page.permissions$, $page.lock$ and $page.wait\_queue$), the new algorithm also tracks the total number of write updates applied to a page ($page.write\_count$), the number of locally generated write updates without an intervening remote update ($page.write\_run\_count$), whether the page is in the multiple-reader single-writer mode described in Section 8.7 ($page.mrsw\_mode$), and a queue of pending stores ($page.store\_queue$).

**fault**($page$, $is\_write$, $is\_rse$): handles a page fault on a DSM page

**if** $is\_write$ **and not** $is\_rse$ **and** $page.write\_run\_count < invalidate\_threshold$
**then**
  disassemble instruction → $operation$, $size$, $is\_release$
  read source register → $write\_value$
  **if** $operation$ = **store then**
    emulate_store_operation($address$, $size$, $is\_release$, $write\_value$)
    **return**
  **else if** $operation$ = **cmpxchg** or $operation$ = **xchg or** $operation$ = **fetchadd**
  **then**
    read compare value register → $compare\_value$
    $result$ = emulate_atomic_operation($operation$, $address$, $size$, $is\_release$,
        $write\_value$, $compare\_value$)
    write $result$ to target register
    **return**
  **else**
    {fall through to acquire_page}
  **end if**
**end if**
acquire_page($page$, $is\_write$, $is\_write$)

---

**emulate_store_operation**($address$, $size$, $is\_release$, $value$): emulates a store

**if** $page.mrsw\_mode$ **and** $page.owner! = node\_id$ **then**
  acquire_page($page$, true, false)
**end if**
store($address$, $size$, $value$)  {perform store locally}
queue_remote_store($address$, $size$, $is\_release$, $value$))

---

**emulate_atomic_operation**($operation$, $address$, $size$, $is\_release$, $new\_value$, $compare\_value$): emulates an atomic operation (**cmpxchg**, **xchg** or **fetchadd**)

**if** $page.owner$ != $node\_id$ **then**
  acquire_page($page$, true, false)
**end if**

**if not** $page.mrsw\_mode$ **then**
  broadcast **mrsw_mode**($this\_node$, $page$)
  wait for all **mrsw_mode_reply** responses
  $page.mrsw\_mode = true$
**end if**
$old\_value$ = operation($address$, $size$, $new\_value$, $compare\_value$)
        {perform operation locally}
**if** $operation$ == **cmpxchg and** $old\_value$ != $compare\_value$ **then**
  {**cmpxchg** aborted}
  **return** $old\_value$
**end if**
queue_remote_store($address$, $size$, $is\_release$, $new\_value$)
**return** $old\_value$

---

**queue_remote_store**($address$, $size$, $is\_release$, $value$): queue a store to be sent to other nodes

---

page($address$).$write\_count$++
page($address$).$write\_run\_count$++
add **remote_store**($address$, $size$, $value$) message to write buffer
**if** write buffer full **then**
  flush write buffer (broadcast and clear)
**else if** $is\_release$ **then**
  schedule expedited flush of write buffer, after configurable delay
**end if**

---

**mrsw_mode**($requesting\_node$, $page$): server for **mrsw_mode** messages

---

**if** writes to $page$ in write buffer **then**
  flush write buffer
**end if**
$page.mrsw\_mode$ = true
send **mrsw_mode_reply**($page$) to $requesting\_node$

---

**remote_store**($address$, $size$, $value$): server for **remote_store** messages

---

page($address$).$write\_count$++
page($address$).$write\_run\_count = 0$

**if** page($address$).$lock$ **then**
  queue store on $page.store\_queue$
**else**
  store($address$, $size$, $value$) {perform store locally now}
**end if**

---

**acquire_page**($page$, $request\_ownership$, $exclusive$): fetches a DSM page

---

**if** writes to $page$ in write buffer **then**
  flush write buffer
**end if**
**if** $page.owner = this\_node$ **then**
  send **invalidate**($page$) to nodes in $page.copyset$
  $page.copyset = \emptyset$
  $page.permissions =$ readwrite
  $page.mrsw\_mode =$ false
  **return**
**end if**
$page.lock = 1$
**if** $manager(page) = this\_node$ **then**
  send **fetch**($this\_node$, $page$, true, $exclusive$) to $page.owner$
**else**
  send **fetch**($this\_node$, $page$, $request\_ownership$, $exclusive$) to $manager(page)$
**end if**
wait while $page.lock = 1$

---

**fetch**($requesting\_node$, $page$, $request\_ownership$, $exclusive$): server for **fetch** messages

---

**if** writes to $page$ in write buffer **or** release seen since last flush (see Section 8.8) **then**
  flush write buffer
**end if**
**if** $page.lock$ **then**
  enqueue message on $page.wait\_queue$
**else if** no forward progress since $page$ received **then**

```
    enqueue message on page.wait_queue
    schedule callback to process page.wait_queue
  else if page.owner = this_node then
    if exclusive then
      page.mrsw_mode = false
    end if
    if requesting_node ∉ page.copyset then
      send page data to requesting_node
    end if
    send fetch_reply(this_node, page, request_ownership, exclusive, page.copyset,
                     page.write_count, page.mrsw_mode) to requesting_node
    if request_ownership then
      page.owner = requesting_node
    else
      page.copyset = page.copyset ∪ {requesting_node}
    end if
    page.permissions = exclusive ? nil : read
    page.write_run_count = 0
  else
    send fetch(requesting_node, page, request_ownership, exclusive) to page.owner
  end if
```

---

**fetch_reply**($from\_node$, *page*, $grant\_ownership$, $exclusive$, *copyset*, $write\_count$, $mrsw\_mode$): server for **fetch_reply** messages

---

```
  if exclusive then
    page.owner = this_node
    copyset = copyset \ {this_node}
    if copyset ≠ ∅ then
      send invalidate(page) to nodes in copyset
    end if
    page.copyset = ∅
  else if grant_ownership then
    page.owner = this_node
```

$page.copyset = (copyset \setminus \{this\_node\}) \cup \{from\_node\}$
**end if**
$page.permissions = exclusive$ ? readwrite : read
$page.mrsw\_mode = mrsw\_mode$
$page.write\_run\_count = 0$
$page.lock = 0$
**if** page data received with this message **then**
drop stores in $page.store\_queue$ from $from\_node$
drop stores in $page.store\_queue$ except last $page.write\_count - write\_count$
(as per Section 8.5)
**end if**
process $page.store\_queue$
process $page.wait\_queue$

---

**invalidate**($page$): server for **invalidate** messages

---

**if** no forward progress since $page$ received **then**
enqueue message on $page.wait\_queue$
schedule callback to process $page.wait\_queue$
**else**
$page.permissions =$ nil
$page.mrsw\_mode =$ false
$page.write\_run\_count = 0$
**end if**

# Chapter 9

# Other infrastructure

In addition to the distributed shared memory system that was the focus of this part of the thesis, there are a number of other components that make the distributed hypervisor a reality. These pieces include: efficient inter-node communication protocols, a mechanism for interprocessor interrupts, an appropriate set of network-transparent virtual devices, and a bootstrap process to co-ordinate the nodes during startup.

## 9.1 Efficient inter-node communication

A vNUMA cluster consists of commodity workstations connected by Gigabit Ethernet. Thus, an obvious requirement is that each node have a Gigabit Ethernet card, and vNUMA must have an efficient driver for that card that can be leveraged for inter-node communication. In many modern virtual machine monitors — including Xen, VMware Server and Microsoft Virtual Server — network device drivers are contained not in the VMM itself but in one of the virtual machines. This not only prevents that particular virtual machine from being distributed, but also impacts latency. As will be demonstrated in Chapter 12, latency is absolutely critical to DSM performance, and this would put such hypervisor designs at a disadvantage. vNUMA contains, within the hypervisor, latency-optimised drivers for a number of Gigabit Ethernet chipsets.

Just as it was sought to achieve the lowest possible virtualisation overhead by developing a thin standalone hypervisor, so it was sought to provide the lowest

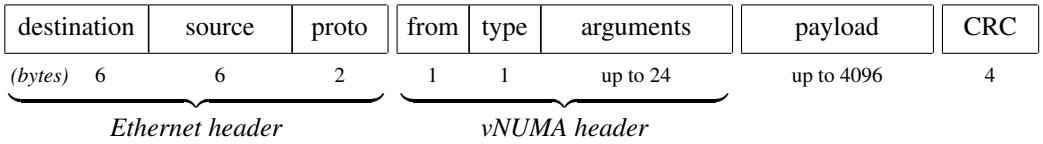



**Figure 9.1:** Anatomy of a vNUMA packet

| Type | Purpose |
|---|---|
| **fetch**[a] | Core DSM protocol (§7.6) |
| **invalidate**[a] | Core DSM protocol (§7.6) |
| **store**[b] | Write-update protocol (§8.1) |
| **ipi** | Inter-processor interrupt (§9.2) |
| **purge**[b] | Global TLB purge (§9.2) |
| **diskio** | Disk I/O (§9.3) |
| **discover**[ab] | Initialisation (§9.4) |
| **configure**[a] | Initialisation (§9.4) |
| **startup** | Initialisation (§9.4) |
| **reset**[b] | Reboots all nodes |

[a]Request/reply pair
[b]Broadcast message

**Table 9.1:** vNUMA message types

possible communication overhead by using a very simple protocol at the Ethernet layer. This avoids the complexity and overheads of protocols such as TCP/IP, which have primarily been designed for wide area networks, and allows the protocol to be customised to the desired requirements.

### Packet format

The format of a vNUMA packet is shown in Figure 9.1. It starts with the Ethernet-mandated header, containing a destination Ethernet address, a source Ethernet address, and a protocol number (which in this case identifies the vNUMA protocol). This is followed by a vNUMA header, specifying the logical source node, message type (Table 9.1) and arguments specific to that message type (such as the page number and request sequence number). Finally there is a variable length data payload and a CRC checksum (mandated by Ethernet). Typically, the

data payload contains page data, although it is also used for other purposes such as transferring write updates in the write-update protocol.

Nominally, the maximum Ethernet payload is 1500 bytes, excluding Ethernet header and CRC; hence a single 4096 byte page requires three frames to transmit. However, almost all Gigabit Ethernet network cards (and some but not all switches) support so-called *jumbo frames*, which allows the maximum frame size to be raised as high as 9000 bytes, easily accommodating an entire page in one packet. vNUMA can either use jumbo frames or it can fragment a page between an arbitrary number of packets. The effect of this is investigated in section 12.2.7. In fact, with low-cost store-and-forward switches, fragmenting the page into multiple frames can be advantageous for latency.

All Gigabit Ethernet network cards support coalescing data from multiple buffers into a single packet, and one can leverage this to avoid copying large amounts of data when constructing a frame. Nevertheless, there is overhead for each additional buffer, so only two buffers are used. The Ethernet header and vNUMA header are constructed in the first buffer; the second buffer references the source page data directly, avoiding any copying in the common case.

If jumbo frames are used then zero-copy receive is also possible using a similar setup, although this has not yet been implemented. Since the headers are of a fixed length, the card can be instructed to receive the headers into one buffer and the data payload onto a separate page (alternatively, a single buffer can be set up crossing two pages in the appropriate way). Then, virtual memory tricks can be used to swap the received page with the target page, without copying. The vNUMA memory management infrastructure makes this relatively simple, although modifications to the network card drivers would be needed to set up the buffers appropriately.

A distinctly problematic case occurs when a page is queued to be transmitted and a local write to the page is intercepted[7]. It would not be safe to resume local execution without applying a local write. However, it is also not safe to apply the write in-place, since it is not known whether the network card has already read the page data or not, and so the copy being sent would be in an indeterminate

[7]Note that it is not necessary to suppress remote writes received during this period; these will be re-applied at the destination and it does not matter if they are applied twice.

state. When this occurs, the page data is copied to a newly allocated frame; this newly allocated frame is swapped into the virtual machine while the network card continues to transmits the old frame. Thus, sending is still zero-copy in the common case, but a single copy is occasionally necessary.

### Ethernet reliability and ordering properties

vNUMA's distributed shared memory system depends, to varying degrees, on the inter-node communication layer providing four properties:

- *Reliability*: no messages are lost or corrupted
- *Ordered delivery*: messages from a single sender arrive at any given receiver in the same order they were sent; if P1 transmits $m_1$ before $m_2$, then P2 must receive $m_1$ before $m_2$.
- *Causally ordered delivery*: messages that are causally related (one message was transmitted following transmission or receipt of another message) arrive in the proper order at any given receiver. This is a superset of ordered delivery that applies to more than two nodes. For example, if P1 broadcasts $m_1$, and then P2 transmits $m_2$ upon receiving $m_1$, then $m_1$ must arrive before $m_2$ (Figure 9.2).
- *Sender-oblivious total order broadcast*: if P1 broadcasts $m_1$, and P2 broadcasts $m_2$, then either all other observers observe $m_1$ before $m_2$, or all other observers observe $m_2$ before $m_1$. Note that in an Ethernet-like network, P1 and P2 do not receive their own broadcasts and cannot make any conclusions as to the total order; this is what is meant by sender-oblivious.

Most distributed systems rely on high-level network protocols to provide these guarantees. However, the vNUMA system is a well-controlled hardware environment. Clearly, if it can be shown that the system hardware exhibits certain properties by design, then it is most efficient to leverage those properties and reduce the software overhead.

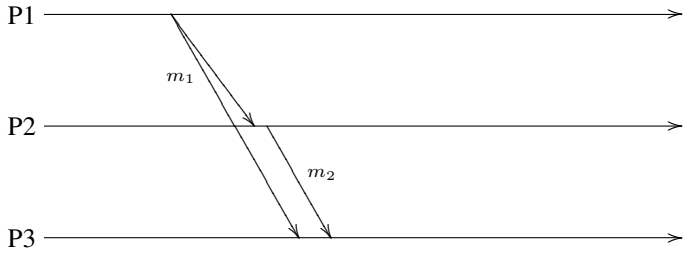



**Figure 9.2:** Causally ordered delivery. $m_1$ is received at P2 before $m_2$ is sent, therefore $m_1$ must be received at P3 before $m_2$ is received.

To understand message delivery in an Ethernet network, it is necessary to consider how a typical Ethernet switch is designed. The majority of commodity switches employ a single switch chip which performs packet processing, as in Figure 9.3; some modern designs are capable of supporting up to 48 ports with the one switch chip.

A switch chip has a transmit and receive process for each port, a switching process, and packet data memory, as shown in Figure 9.4. When a packet is incoming, the receive process first transfers the data into the packet data memory. The switching process is then notified; it determines the set of output ports based on the destination Ethernet address of the packet, and adds a reference to the packet in each of the corresponding output queues. Each output port's transmit process gradually transmits packets from its queue in FIFO (first-in first-out) order. When no references to the packet remain in any output queue, the packet's buffers are released.

Ordered delivery is a natural consequence of such a design. If $m_1$ is transmitted by P1 before $m_2$, then $m_1$ is received first, switched first, queued to the output FIFO first, and transmitted on the output port first. Causally ordered delivery is also guaranteed: if P2 has received $m_1$, then any other recipients' output queues must also already contain $m_1$; thus an observer will necessarily receive $m_2$ after $m_1$. In fact, total order broadcast is also provided by the atomicity

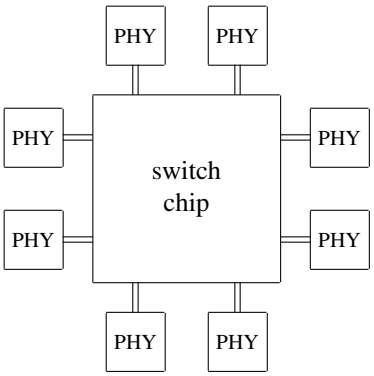



**Figure 9.3:** Typical design of a Gigabit Ethernet switch. Each PHY transceiver is connected to a port.

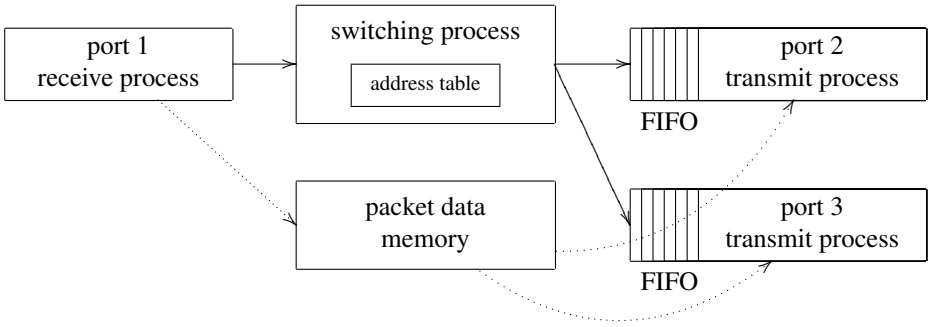



**Figure 9.4:** Basic operation of a switch chip

of the switching process: either the switching process first considers $m_1$ and queues it to all output queues before $m_2$, or it first considers $m_2$ and queues it to all output queues before $m_1$.

It should be noted that some switches support *cut-through* switching, in which case transmission on the output port may commence immediately after the destination address is seen — and while the rest of the packet is still being received — as opposed to *store-and-forward* switching which first requires the whole packet to be received. Since cut-through switching is only used when the output queue is empty, there is no practical distinction to be made in terms of the ordering guarantees provided. Indeed cut-through switching is very much desirable for a vNUMA system, since it reduces latency.

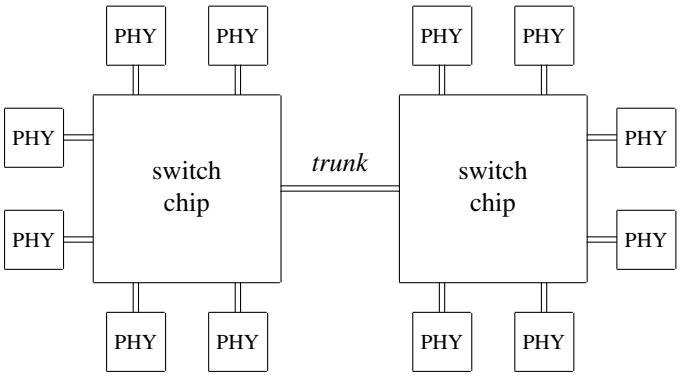



**Figure 9.5:** Design of a switch with two switch chips

A more complicated situation arises if the Ethernet fabric has more than one switch chip, such that there is no longer a single central point at which switching occurs. This occurs in some large switch designs, or if multiple switches are connected together. An example topology with two switch chips is shown in Figure 9.5; the set of ports is now divided into two groups, connected together with a trunk that is essentially like another port.

Assuming that packets are forwarded over the trunk in FIFO order, it is possible to show that such a topology still guarantees ordered delivery and causally ordered delivery, even for nodes located in different groups: these properties hold between a source node and the trunk, they hold between the trunk and the destination node, and they are transitive. Total order broadcast, however, *cannot* be guaranteed in this scenario: a broadcast will first be queued to local ports before being forwarded over the link. There is no longer a central atomic switching process to provide a total order. Thus, in such a topology, one could not rely on atomic broadcast; protocol-level solutions become necessary. Firstly, the algorithm of 8.5 would need to use the more complex vector clock solution rather than the simpler algorithm described. More fundamentally, remote store atomicity would be violated; the Itanium architecture requires remote store atomicity for stores with release annotations. However, since causality and coherence are still preserved, it is unlikely that this would affect many algorithms in practice. This could be resolved with a more complex protocol for store atomicity, as has been

done in this thesis for coherence.

The vNUMA protocol is also heavily optimised for the case where there is no packet loss, although some consideration has been given to detection and recovery. It can reasonably be assumed that link-level transmission over short distances of quality cabling is reliable — since all physical parameters are well within their specifications, signal-to-noise ratio is high, and Gigabit Ethernet is capable of correcting a certain number of random bit errors. Thus, packets are only lost in two cases: if the receiver has insufficient buffers, or the switch has insufficient buffers. Receive buffers are within the control of vNUMA, and indeed can be set high enough that no loss occurs even at network saturation. Switch buffers also tend to be generous, but for certain pathological communication patterns it is nonetheless possible to queue data faster than the available output bandwidth.

Dealing with packet loss in the base DSM protocol is relatively simple. The base protocol uses request/reply pairs, and the request is retransmitted after a timeout. The sequence number contained in the packet allows a receiver to detect a request that it has already processed (implying that the reply was lost). Further, the request/reply nature of the protocol bounds the number of in-flight messages and the required buffer space in the switch. The greatest amount of buffer space is taken when each node is waiting on page data and that page data is in-flight; for $N$ nodes, this is a little over $N$ multiples of the page size, or 32 KiB for 8 nodes — significantly smaller than the packet buffers in a typical 8 port switch (at least 128 KiB). Additionally, if jumbo frames are *not* used, then this requirement is reduced, since the page data moves through the switch in a pipelined fashion.

In the case of the write-broadcast protocol, the situation is more complicated, since there are no synchronous replies. A loss is detected when the next packet from the same sender arrives, with an unexpected sequence number, and at that point the earlier update could be re-requested. This does not violate coherence, since the coherence algorithm can cope with receiving writes in any order. It is also processor consistent, assuming that the earlier update is applied before the later update from the same sender. However, the relative order of writes from different processors is affected; globally it can violate remote store atomicity and causality. This would not be a problem for the IA-32 architecture, which only guarantees processor consistency; however as previously mentioned the Itanium

architecture insists on remote store atomicity for the subset of stores with release annotations. A more complex protocol that would provide this level of store atomicity in the face of lost packets is theoretically possible but beyond the scope of this thesis.

## 9.2 Inter-processor interrupts

In addition to providing shared memory, vNUMA must also provide *inter-processor interrupts* (IPIs), which allow an operating system to send a notification to another processor. In the Itanium architecture, IPIs are sent by writing a value to a virtual array located at a certain address; the index in the array specifies the destination processor and the value specifies the interrupt number to deliver. vNUMA intercepts these writes via the same mechanism as used by the write-update protocol, and sends an IPI message to the destination node which delivers the interrupt[8].

The Itanium architecture also provides a global TLB purge instruction, which allows an entry to be invalidated from remote TLBs without needing to invoke OS code on every processor. This is implemented in the obvious way, by broadcasting a TLB purge request to all processors. The global TLB purge instruction has release semantics, so local write buffers must be flushed before the TLB purge message is sent.

## 9.3 Distributing I/O

vNUMA also must provide certain virtual devices to the operating system. As presented in Part I, the operating system interface provides for three virtual device classes: network (Ethernet), disk (SCSI) and console.

In that chapter it was assumed that these facilities could simply be mapped directly onto physical devices. However, in a distributed environment this is more complicated. Linux assumes that any device can be accessed from any

[8]While a separate message type is used in vNUMA, it would also be possible to use a normal write message for this purpose; a node that receives a write to its own special interrupt address then delivers an interrupt.

node. However, in a vNUMA cluster, there will be multiple physical devices of each type, each of which can only be accessed on a particular node. For each device type, there are two possible options: either (a) to present each physical device to the operating system, and route non-local I/O requests to the node with the requested device; or (b) to present a single virtual device that vNUMA is responsible for mapping onto one or more physical devices.

### Network

The network device type lends itself to solution (b): a single virtual Ethernet card. Since processes arbitrarily and transparently migrate between nodes, and TCP/IP connections are fixed to a certain IP address, it is necessary to have a single IP address for the cluster. With a single virtual Ethernet interface, the user can simply configure an IP address on that interface; it would be confusing and fruitless to provide multiple virtual interfaces.

Since all of the nodes are on a shared Ethernet, outgoing packets can be sent from any node. The design of Linux is such that packets are generally sent from whichever node the sending process is running on, which is the optimal solution. In order to avoid confusing network switches (which keep track of which port an Ethernet address has been seen on), vNUMA substitutes the Ethernet address of the real local network card into outgoing packets.

Unfortunately, due to the limitations of the IP protocol, there is no easy way to avoid all incoming packets arriving at a single node (the node with the specific Ethernet address that appears in Address Resolution Protocol replies). The current implementation simply delivers the packets on that node. This has the advantage that the receiving part of the driver and network stack always runs on a single node, but the disadvantage that the actual consumer of the data may well be running on a different node. An alternative implementation could be to use a dedicated load balancer in between the cluster and the outside world. This load balancer could distribute packets to different nodes based on the TCP/UDP port numbers, which provide a hint as to where different applications are running.

### Disk

In such a system the ideal method for dealing with disks is to connect them to a storage area network, so that they can be accessed from any of the nodes. For example, if the disks were connected to the Ethernet, a protocol like iSCSI could be used. Virtual Iron's VFe hypervisor relies on such a storage area network topology.

However, storage area networks are not widely deployed outside the enterprise, and the vNUMA philosophy is to support commodity hardware. Thus in the absence of a SAN, vNUMA provides the virtual machine with a single virtual SCSI disk. vNUMA routes accesses to this disk back to the bootstrap node, which is assumed to be a node with physical disks. Given a disk request — which can contain a scatter-gather list — the bootstrap node acquires the required pages via the DSM system, performs the I/O, and delivers the completion interrupt locally. As in the Ethernet case, the SCSI interrupt handler always runs on that one node, even though the ultimate consumer may be remote.

Clearly having one node responsible for all disk I/O can become a bottleneck; an alternative possibility would be to employ striping or mirroring across available disks on other nodes.

### Console

The console device is primarily used for debugging; it is expected that most users would access a vNUMA system via the network. Hence the current implementation sends all console output to the local console, which has the strange but informative (for developers) effect that output migrates as processes migrate, while input can be accepted on any node. An alternative implementation might direct output to the bootstrap node. In this case a timed coalescing algorithm would be necessary, to avoid a multitude of single character packets flooding the network.

## 9.4 Bootstrap process

Finally, the process by which a vNUMA system is booted deserves a mention. All of the nodes in the cluster must be configured to boot the vNUMA hypervisor image in place of an operating system kernel. Then, one of the nodes is selected by the administrator to be the bootstrap node, by providing it with a guest kernel image and boot parameters; the other nodes need no special configuration.

Once the bootstrap node initialises, it broadcasts discovery messages on the network to locate the other nodes that are online and determine their resources. It then sends a configure message to each node that will participate in the vNUMA system, providing information necessary for communication with other nodes: that node's logical number in the cluster, the total number of nodes, and a table of MAC addresses of the other nodes.

The bootstrap node then copies the guest kernel image into memory allocated for it, and commences booting it, no differently from a uniprocessor virtual machine. The other nodes await their turn. Eventually the guest kernel registers an SMP startup address and starts sending startup IPIs to wake up other processors. In vNUMA, this sends startup messages over the network; those nodes then blindly start executing at the given address. Immediately they encounter a page fault, which is handled by the DSM system by fetching the appropriate code page from the bootstrap node, and in this way — page by page — these remote processors come alive.

# Part III

# Evaluation

The goals of this evaluation are to:

- demonstrate that the vNUMA system works for a variety of applications,
- determine the performance that can be achieved,
- demonstrate the effects of optimisations described in this thesis,
- characterise remaining overheads,
- compare vNUMA with a real SMP or ccNUMA system, and
- compare vNUMA with a userspace DSM system or other application-specific solutions.

Three particular types of application are considered, which cover some of the most common use scenarios for large computer systems: computationally-intensive scientific workloads, software build workloads, and database server workloads. Chapter 10 will present information about the test environment and methodology. Chapter 11 will introduce the benchmarks, with results that show the performance achievable on vNUMA. Chapter 12 will explore implementation trade-offs behind these results, and how specific decisions affect performance.

# Chapter 10

# Methodology

## 10.1 Test environment

The test system used for this experimental work consists of HP rx2600 servers with 900MHz Itanium 2 processors, connected using Gigabit Ethernet via an HP ProCurve 2708 switch. Most analysis was done on a four-node cluster, but four additional nodes were added for scalability testing.

The current implementation of vNUMA, as described in this thesis, is around 10,000 lines of code. Of this around 4000 lines constitute a generic Itanium virtual machine monitor, the DSM system is around 3000 lines of code, and the remainder deals with machine-specific initialisation and fault handling. In addition to this core which was developed from scratch, C library functions and device drivers from the Kenge project [76] were utilised on an as-needed basis.

Since vNUMA does not yet support SMP within a node, only one CPU was used in each server; the terms CPU, processor and node are used interchangably. In a production system, it would obviously be preferable to utilise all local resources before going beyond the node boundary, particularly with the increasing prevalence of multi-core processors. However, the intent of these benchmarks is to measure the achievable inter-node performance, so additional hierarchy would only serve to complicate the evaluation.

Linux 2.6.16 was used as the operating-system kernel, both on vNUMA and for comparative measurements. Default configuration settings were used where possible, including a 16 KiB page size. This does provide the native kernel

with a certain advantage, since vNUMA normally uses 4 KiB hardware pages, and therefore incurs more TLB misses than a native kernel using 16 KiB page size. On the other hand, having the guest kernel compiled with 16 KiB page size also benefits vNUMA, since virtualisation overhead is reduced by the larger page size at the interface between the hypervisor and guest kernel. Overall, it was considered that configuring both kernels with 16 KiB page size provides the most sensible comparison. (The one exception is the Treadmarks measurements; these were performed with 4 KiB page size to provide a fair comparison of DSM performance.)

The pre-virtualisation technique, described in Chapter 4, was used to transform the Linux kernel for efficient execution on vNUMA. Three minor changes were made manually. Firstly, the Linux `wait_on_bit_lock` function was modified as described in Section 8.9. Secondly, the `clear_page` function was replaced with a hypervisor call to allow it to be implemented more optimally. Since it is known that the existing version of the page is no longer needed, there is no need to fetch the previous data; the page can be invalidated asynchronously. Finally, the kernel linker script was modified to place the `.data.read_mostly` section — which contains data that is rarely written and should be read-shared — on a separate page. Otherwise, it happens to share a page with the `.data.cacheline_aligned` section, which is generally undesirable (the latter contains locks and data that is written to frequently).

Wherever possible, results presented are a median of the results from at least ten runs of a benchmark. The median was chosen as it naturally avoids counting outliers.

## 10.2 Analysis tools

vNUMA contains instrumentation to allow comprehensive analysis of overheads. Profiling can be started and stopped to delineate the particular benchmark under test. During the profiled period, a number of performance metrics are collected. Firstly, the total time is measured using the Itanium cycle counter, and the time spent in the hypervisor is measured using one of the counters in the Itanium performance monitoring unit (configured to count the number of cycles that the

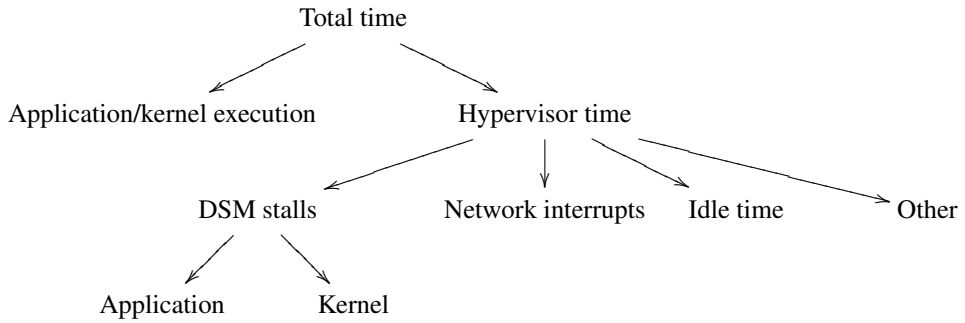



**Figure 10.1:** Breaking down execution time in vNUMA

processor is executing at the most privileged level); using the hardware counters for this ensures that nothing is missed. Then, as a subset of this hypervisor time, specific regions within the hypervisor are timed. This includes stalls waiting for DSM communication, network interrupts (representing the overhead of processing requests and notices from other nodes) and idle time (when the guest kernel has no runnable processes and enters a sleep state). The DSM stalls are further classified into those originating from the guest kernel, and those originating from the guest user application.

The effect is that the vNUMA provides a hierarchical breakdown of the total time, as shown in Figure 10.1. This data is provided separately for each node, but the values can be summed across the nodes to account for all of the available processor time. This provides an effective overview of where time is being spent.

Typically, the greatest vNUMA overheads lie in DSM stalls. A DSM stall occurs as a result of a faulting memory access that invokes the DSM system, when the fault cannot be resolved without waiting on network communication. This typically occurs because page data is not available to satisfy a read, or because an atomic read-modify-write instruction must acquire exclusive ownership of the page. In some cases — such as if the page is in a single-writer mode — ordinary write instructions may also stall waiting on ownership, but generally write stalls can be avoided by buffering writes using the mechanisms described in Chapter 8.

To allow analysis of the DSM stalls that occur during the runtime of an application, data about individual stalls is logged to a profiling buffer. Each time

the buffer becomes full, the data is sent to a central profiling node, which can be any computer on the same network as the vNUMA cluster. The data collected includes:

- the node that faulted,
- the virtual address that was accessed,
- the underlying DSM page number,
- the address of the referencing instruction,
- the type of the access (read or write; instruction, data or register stack),
- the node(s) that were involved in resolving the fault,
- whether the response included new page data, or whether the cached copy was found to be up-to-date,
- the number of outgoing frames in the network card's transmit queue that had to be waited for,
- the relative time at which the fault occurred, and
- the time taken to resolve the fault.

Based on this foundation, a variety of analysis tools can be constructed to analyse the performance of the vNUMA DSM. The output of these tools will be presented throughout this evaluation.

# Chapter 11

# Benchmarks

## 11.1 HPC benchmarks

One of the most exciting applications of powerful computer systems is the numerical solution of computationally complex problems, such as those encountered in the physical sciences and engineering. Simulating the gravitational interaction of planets, the air flow over an aeroplane wing, or the deposition of metal atoms on a silicon wafer — to name but a few examples — all require complex numerical calculations. The application of large computers and clusters of computers to solve such problems is generally termed *high-performance computing* (HPC).

Since HPC applications are computationally intensive, they are usually designed to make use of multiple processors in parallel. This means that they are generally written to one of three paradigms: explicitly passing data between processing nodes (often via middleware such as MPI [69]), using a middleware system that provides distributed shared memory, or using multiple operating-system threads and relying on hardware support for shared memory. vNUMA can provide benefits in all of these scenarios: it can transparently distribute a program written for a shared memory system, or it can be used in place of a middleware DSM system (the benefits will become evident in the upcoming chapters), or it can even be used to host applications that use explicit message passing. In all cases it provides the advantages of a single system image — appearing to the user as a single node, with all the conveniences of a single process space and a single filesystem to store inputs and outputs.

In theory, explicit message passing can always achieve higher performance than shared memory, since shared memory is a higher-level abstraction on top of message passing. Thus one might question the usefulness of shared memory systems for high-performance applications. Nonetheless shared memory remains a compelling and widespread programming model, due to its simplicity. The ubiquitousness of the shared memory abstraction is evidenced by the fact that the great majority of SMP systems do not provide explicit message passing between processors, only shared memory (despite the fact that the physical interconnect is message-based!).

The main question, then, is whether vNUMA can provide adequate performance for HPC applications. The TreadMarks DSM system [50], as introduced in Chapter 6, was used for comparison. TreadMarks is a second-generation DSM system that achieves good performance using a release consistency model. While TreadMarks may no longer represent the state of the art in DSM research, it is one of the few DSM systems that has been widely used in the scientific community. This is because it is readily obtainable, relatively easy to use, and stable — attributes that set it apart from many research systems.

TreadMarks is distributed with an assortment of benchmark applications which have already been ported to TreadMarks, mostly from the SPLASH-2 benchmark suite from Stanford University [91] and the NAS Parallel Benchmarks from NASA [28]. These benchmark applications are well-known in the field and representative of a wide set of HPC problems.

The versions of the applications that are provided with TreadMarks explicitly make use of TreadMarks library functions, as is required by the TreadMarks programming model. To allow these to run on an SMP system, ccNUMA system, or on vNUMA, a stub library was created that implements the TreadMarks functions in terms of `fork()` and shared memory. Barriers and locks were implemented using atomic operations on the shared memory, as they might normally be realised on an SMP system. This allowed running the same application code for all of the configurations, simply by substituting different libraries. Use of the applications provided with TreadMarks (and optimised for TreadMarks) also ensured that TreadMarks was not unfairly disadvantaged.

### Benchmark descriptions

For this analysis, the C language subset of the benchmarks was chosen; the author is not well versed in FORTRAN and the C language benchmarks are sufficient to cover a wide range of different memory access patterns. The benchmarks used are:

- **Barnes** [91, 62]: Simulates the gravitational interaction of many particles using the hierarchical Barnes-Hut algorithm. While the particle data is in a contiguous array, the particular particles accessed and updated by a processor are data-dependent, reflecting the relative position of each particle. Thus **Barnes** exhibits sparse read and write patterns and the potential for false sharing.

- **CG** (conjugate gradient) [28]: Approximates one eigenvalue of a very large sparse matrix, using the inverse power method. The matrix is too large to be stored as an explicit array; instead it is stored in terms of the non-zero values and their locations, the list of which is partitioned between processors. The inverse power method is an iterative method, each step of which involves dividing a vector by the matrix (in other words, solving the equivalent linear system $Ax = b$). The particular method used to solve the linear system (the conjugate gradient method) is itself an iterative method, each step of which involves multiplication of the matrix by a vector and calculation of gradients between resulting vectors. Thus at its core **CG** consists of a large number of distributed matrix-vector operations; in each of these a vector is distributed to processors and multiplied by the local portion of the large matrix.

- **FFT** [28, 62]: Performs a fast fourier transform in three dimensions. The 3D space is partitioned between processors in one of the dimensions, such that each processor has a subset of planes. FFTs are first calculated within each processor's local planes, then the matrix is transposed, and the final FFT is performed. Significant communication occurs in the transposition step, in which each processor accesses every other processor's planes.

- **Gauss**: Performs Gaussian elimination on a matrix. For each row, the processor assigned to that row chooses a pivot element, swaps it into the

diagonal position, and divides the row by it. After a barrier, each processor fetches the entire pivot row and the index of the pivot column. It performs the appropriate column swap and subtracts a multiple of the pivot row from its own rows.

- **IS** (integer sort) [28, 62]: Sorts a set of integers of known range using a counting sort method. The set of integers is partitioned between processors, and each processor first calculates the frequency distribution of its own integers. Then, the frequency distribution is accumulated into a global frequency distribution; to minimise false sharing, the processors update separate portions of the frequency distribution at each step. Finally, each processor reads back the global frequency distribution to determine the position of each of its integers in the overall ordering.

- **MG** (multigrid) [28]: Computes an approximate solution to a partial differential equation $\nabla^2 u = v$ in three dimensions, using a numerical method that interpolates between grids of different resolution. Like **FFT**, the grids are partitioned between processors in one of the dimensions. Each basic operation — projection, interpolation, residual calculation and smoothing — involves calculating cells in an output grid as a function of the set of cells spatially adjacent to the corresponding cells in an input grid (in each of the three dimensions). Hence, communication is necessary when performing calculations for planes that border those owned by neighbours.

- **Raytrace** [91]: Renders a three-dimensional scene into a two-dimensional image using ray tracing. The scene is represented using various shared data structures that represent objects, lights and so forth, and each processor is assigned a certain number of 16x16 blocks of output pixels that it must calculate. While access to the data structures describing the scene can be very irregular, almost all of these accesses are read-only, so little communication is necessary during most of the execution. At worst there is some false sharing in the output frame buffer to which pixel values are written. Once a processor runs out of work, work is taken from other

processors' work queues; however this only happens near the end of a run and does not contribute significantly to communication overheads.

- **SOR** (successive over-relaxation) [62]: Another linear equation solver, this time using the successive over-relaxation method. The implementation is specifically designed to minimise false sharing. Each row of the matrix has a red and a black half, each of which is page-aligned. Nodes always read red and write black positions and then, after a barrier, read black and write red positions, such that there are never any read/write conflicts.

- **TSP** [62]: Finds a solution to a given instance of the well-known travelling salesman problem, using a branch-and-bound method. Partial tours are placed in a shared priority queue, protected by a lock, which each processor draws from whenever it is idle; this data structure involves irregular read-write sharing. If the next partial tour is close enough to a complete tour for a single processor to complete, the shortest possible completion of that partial tour is found exhaustively by that processor, otherwise the problem is split and the resulting partial tours pushed back onto the priority queue for other processors. Whenever a processor finds a solution better than the current best solution, it updates the global best tour accordingly. Processors frequently access this best tour information to prune less optimal solutions; thus this structure is read frequently by all processors and updated intermittently.

- **Water** [91, 62]: Simulates molecular interactions within a liquid ($H_2O$). The **Water** benchmark is similar in many ways to **Barnes**. Like **Barnes**, it involves calculating forces on the set of particles that are spatially 'near' a particle, and updating those particles appropriately. However, the false sharing in **Water** is not as serious as **Barnes**, since there are far fewer particles per page (just over 6, versus 39 for **Barnes**, although neither enforce page alignment). Also, while **Barnes** both reads and writes particles in an unpredictable order, **Water** reads data contiguously.

The **Water** benchmark was modified in one respect to improve performance: the per-molecule locks were placed together with the data in the molecule array,

rather than maintaining a separate array of locks. This does not violate the specification of the benchmark, and placing locks together with data is standard SMP programming practice; having a contiguous lock array may necessitate wasting a whole cache line (or in vNUMA a page) for each lock to avoid unnecessary contention.

### Results

Figure 11.1 shows an overview of results for each benchmark. While the ultimate limits of scalability are difficult to establish without a much larger cluster, vNUMA was designed for optimal performance on a small cluster. The graphs demonstrate that vNUMA performs very competitively in this environment. On a four or eight node cluster it surpasses TreadMarks on all of the benchmarks. This is particularly true for **Barnes**, **Water**, **TSP** and **IS**.

vNUMA's performance advantages in these benchmarks can be attributed to two factors. Firstly, as a hypervisor-based DSM system vNUMA is not restricted to using operating-system services for memory management and communication, and can therefore achieve much lower latencies. This provides improvements for all benchmarks, but particularly those that are communication-dominated; **IS** is the most extreme case.

Secondly, TreadMarks always uses an invalidation-based protocol (propagating invalidations during barriers and lock acquisitions, and lazily obtaining the actual page changes). The main advantage of an invalidation-based protocol is that it eliminates redundant communication when changed data is not needed by the recipients. However, in the case that the data *is* needed, it makes the protocol particularly sensitive to latency. A subsequent read must synchronously wait for data to be obtained from the writer, and this large transfer of data tends to be high-latency. In contrast, vNUMA uses a write-update protocol by default (while falling back to invalidation if update bandwidth proves excessive). In cases where there is a need for read-write sharing (**Barnes** and **Water** due to false sharing, and **TSP** in which all nodes frequently read a shared data structure which is written to intermittently), vNUMA can achieve significant benefits by propagating updates to readers without invalidating read copies.

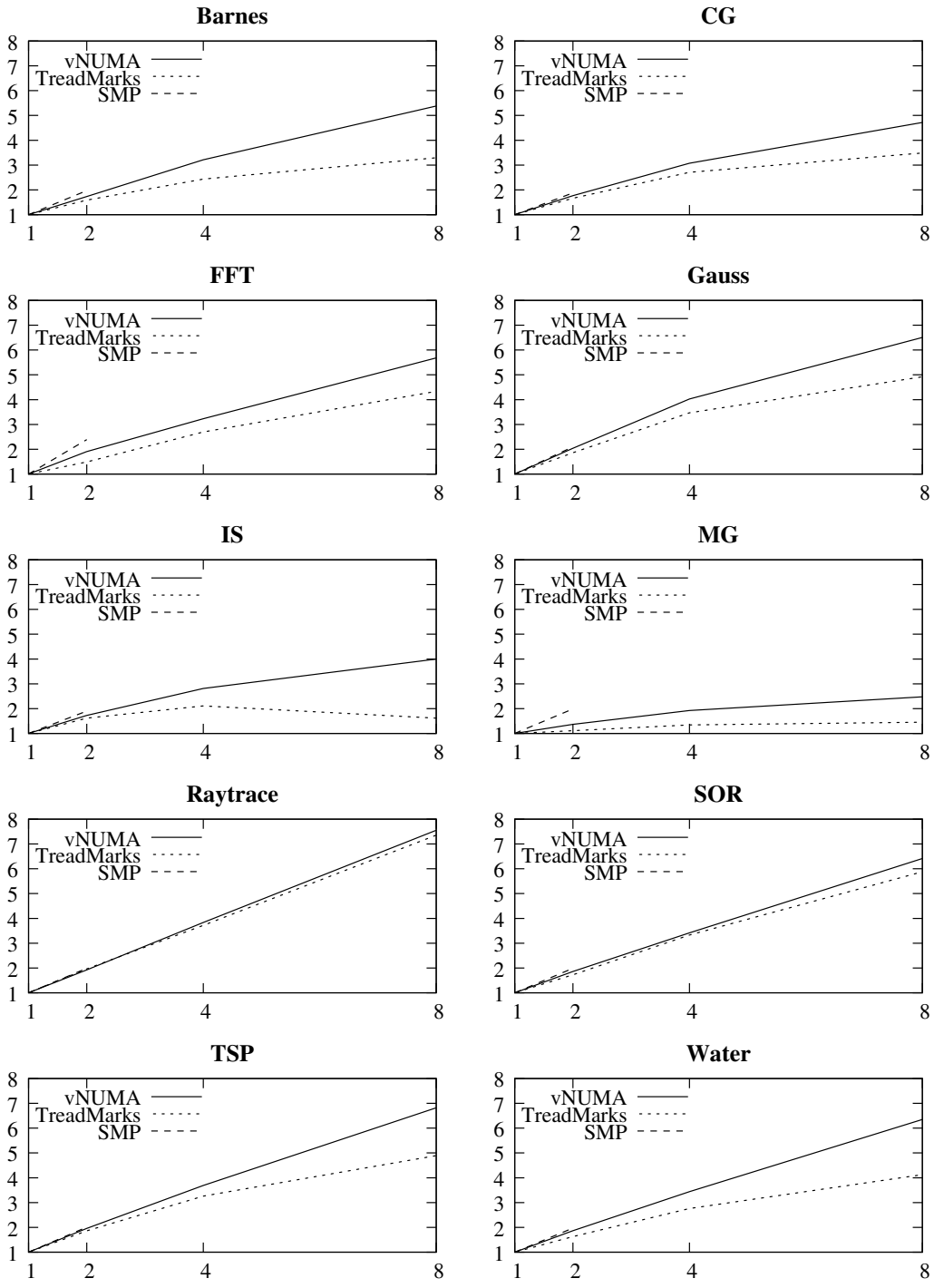



**Figure 11.1:** HPC benchmark performance summary. Horizontal axes represent number of nodes, vertical axes represent speed-up.

**SOR** and **Raytrace** are examples of benchmarks that perform well on TreadMarks as well as vNUMA. **SOR** has no false sharing, with 16 KB blocks of data migrating from one processor to another in each step. Thus, most DSM protocols can provide good performance. Similarly **Raytrace** is almost embarrassingly parallel, with little communication necessary for most of the computation. Although there is some false sharing in the output frame buffer, there are few barriers that would force synchronous communication, and therefore the multiple-writer protocols in both vNUMA and TreadMarks perform well.

**MG** proves the greatest challenge for both TreadMarks and vNUMA. While the actual data resolution uses powers of two, periodic boundary conditions are implemented by replicating boundaries at opposite sides of the grid. This simplifies the program but makes each grid an awkward size in which planes are not page-aligned (66x66x66, 34x34x34, 18x18x18). Additionally, to keep the boundaries in each dimension synchronised, the first and last processor must exchange entire planes after each stage, which requires significant communication. Nonetheless, vNUMA is definitively ahead, achieving a speed-up of 1.92 on four nodes compared to TreadMarks' 1.34.

More detailed analysis of how various factors contribute to performance will be presented in the next chapter, but first the remaining benchmarks will be introduced.

## 11.2 Compile benchmark

A second application where large servers and clusters are commonly deployed is to improve the speed of software builds. Software development cycles often rely on frequent compilation and testing, and therefore developer productivity depends critically on fast compilation times.

While there exists software that can help to distribute builds across a cluster of separate workstations (`distcc` [78]), there are a number of disadvantages of such a model. Firstly, to protect against differences in the available files and environment between cluster nodes, `distcc` only distributes the core C/C++ compilation step; for each file, self-contained (pre-processed) source code is sent over the network and object code is received. Pre-processing and linking stages

are still performed on a single node, as well as compilation of any other languages that may not be easy to incorporate into the `distcc` model. It is also left to the system administrator to ensure that an identical C/C++ compiler is installed on all nodes of the cluster, otherwise there can be subtle build-to-build variations that make debugging difficult.

In contrast, vNUMA provides a single system image model which is no different from an SMP server; there is only one server to deal with, and any tool with multiple processes or threads distributes transparently. If a vNUMA cluster can perform comparably to `distcc`, then this provides compelling advantages.

When run on a disk-backed filesystem, compile benchmark throughput is significantly affected by disk performance. Since I/O performance is not a core goal of this work, this factor was eliminated by running the compile benchmark with all files on a memory-backed filesystem (a *RAM disk*). The traditional choice of compile benchmark — a Linux source tree — is prohibitively large for the workstations' RAM, so the vNUMA source tree was used instead. However, the nature of the workload is similar: a large number of parallel compiles (invoked via make -j*N*) finishing with a link stage which is not parallelisable. Conventional wisdom suggests setting *N* to twice the number of available CPUs; this rule was also followed for this evaluation, and generally produces the best results.

The results are summarised by Figure 11.2. vNUMA performance is practically identical to `distcc`, despite the fact that the two approaches are quite different in nature. The line labelled 'Optimal' is an extrapolation of SMP results, based on an idealised model where the parallelisable portion of the workload (86%) scales perfectly. On 4 nodes, the ideal speed-up is 2.8, while both vNUMA and `distcc` achieve 2.3. On 8 nodes, the ideal speed-up is 4.0, while both vNUMA and `distcc` achieve 3.1.

While the effective performance of both systems is similar, the overheads that limit the performance are different. In the case of `distcc`, the overheads stem from the centralised pre-processing of source files (which creates a bottleneck on the first node), as well as the obvious overheads of transferring all source files and results over the network. In the case of vNUMA, there is overhead from the DSM system and virtualisation. A breakdown of vNUMA's processor time usage on four nodes is presented in Figure 11.3. The idle time represents non-

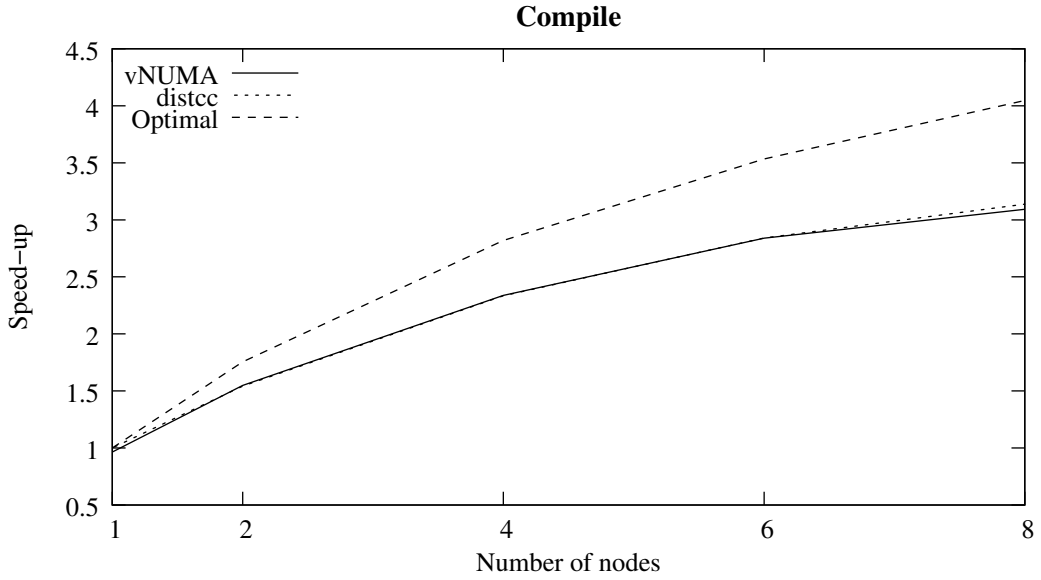



**Figure 11.2:** Compile benchmark performance summary

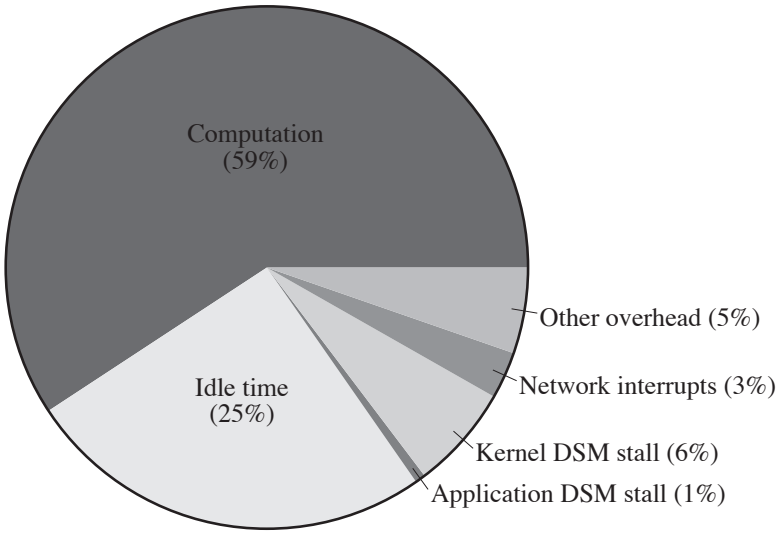



**Figure 11.3:** Time breakdown for compile benchmark (4 nodes)

parallelisable stages of the build, and while high, it is consistent with the expected value from the idealised model. The vNUMA overheads total 15%, with DSM stalls accounting for 7%. The 5% counted as 'other overhead' represents both the cost of intercepting writes ($\approx 3\%$) and other virtualisation overheads.

The majority of the DSM stalls originate from the guest kernel. This is because the compiler processes do not themselves communicate through shared memory. Their code pages are easily replicated throughout the cluster and their data pages become locally owned. However, inputs and outputs are read from and written to the file system, which shifts the burden of communication onto the kernel. In general, the compile benchmark can be considered representative of an application that consists of many processes which do not interact directly but interact through the filesystem.

Profiling the kernel overheads shows that the largest communication costs arise from maintaining the page cache (where cached file data is stored), and acquiring related locks. Similarly the file system directory entry cache (which caches filenames), and related locks, also feature as major contributors. Nonetheless, considering that the overall overhead is no greater than that of `distcc` — a solution specifically crafted for distributed compilation — this seems a small price to pay for the benefits of a single system image.

## 11.3 Database benchmark

Database servers are another application where large SMP and ccNUMA servers are commonly deployed, and commercial database benchmarks such as TPC-C [86] are frequently quoted when such servers are advertised. Most database server software has multiple threads or processes to service requests, and therefore large SMP systems can achieve high throughput. In contrast, distributing a single database across multiple nodes of a workstation cluster is difficult, and most software cannot do so transparently. The question is whether vNUMA can be used for such an application.

The database server used for this evaluation is PostgreSQL [80], one of the two most popular open source database servers used on Linux (the other being MySQL [75]). It was important to use an open source database server so that its source code could be inspected to understand performance problems. For the same reason — ease of understanding — simple synthetic benchmarks were employed instead of a complex hybrid workload such as TPC-C. Two tables were initialised with 10,000 rows each: one describing hypothetical users of a system, and the

other representing posts made by those users on a bulletin board. A pool of client threads then performed continuous queries on this table[1]. The total number of queries completed in 30 seconds (after 5 seconds of warm-up) is recorded. This is similar in principle to benchmarks like TPC-C, but it utilises a smaller number of tables and a simpler mix of transactions.

Five different types of queries were used:

- **SELECT**: retrieves a row from the users table by matching on the primary key
- **SEARCH**: retrieves a row from the users table by searching a column that is not indexed
- **AGGREGATE**: sums all entries in a certain column of the users table
- **COMPLEX**: returns information about the five most prolific posters (this involves aggregating data in the posts table, and then performing a 'join' with the user table)
- **MIXED**: randomly selects one of the above queries to perform

The results are summarised in Figure 11.4. vNUMA performs well for **COMPLEX** and **MIXED**, which involve a base throughput of tens of queries a second. However, performance is degraded for the higher-throughput workloads, **SEARCH** and **AGGREGATE**, and most significantly so for **SELECT**, which involves little computation per query and can thus usually achieve thousands of queries a second on a single node. **SEARCH** and **AGGREGATE** barely manage to regain single-node performance on 8 nodes, while **SELECT** does not scale.

The cause of this throughput-limiting behaviour is simple: using multiple distributed nodes suddenly introduces the potential for much larger communication and synchronisation latencies. If one considers that each query involves at least a certain number of these high-latency events, then the maximum query throughput per node is inversely proportional to the number and cost of those events.

**SELECT** will be analysed further, since it is the workload with the worst performance. A breakdown of processor time usage is shown in Figure 11.5.

[1]The number of client threads was chosen to maximise throughput; the maximum throughput invariably occurred when the number of threads equalled the number of processors.

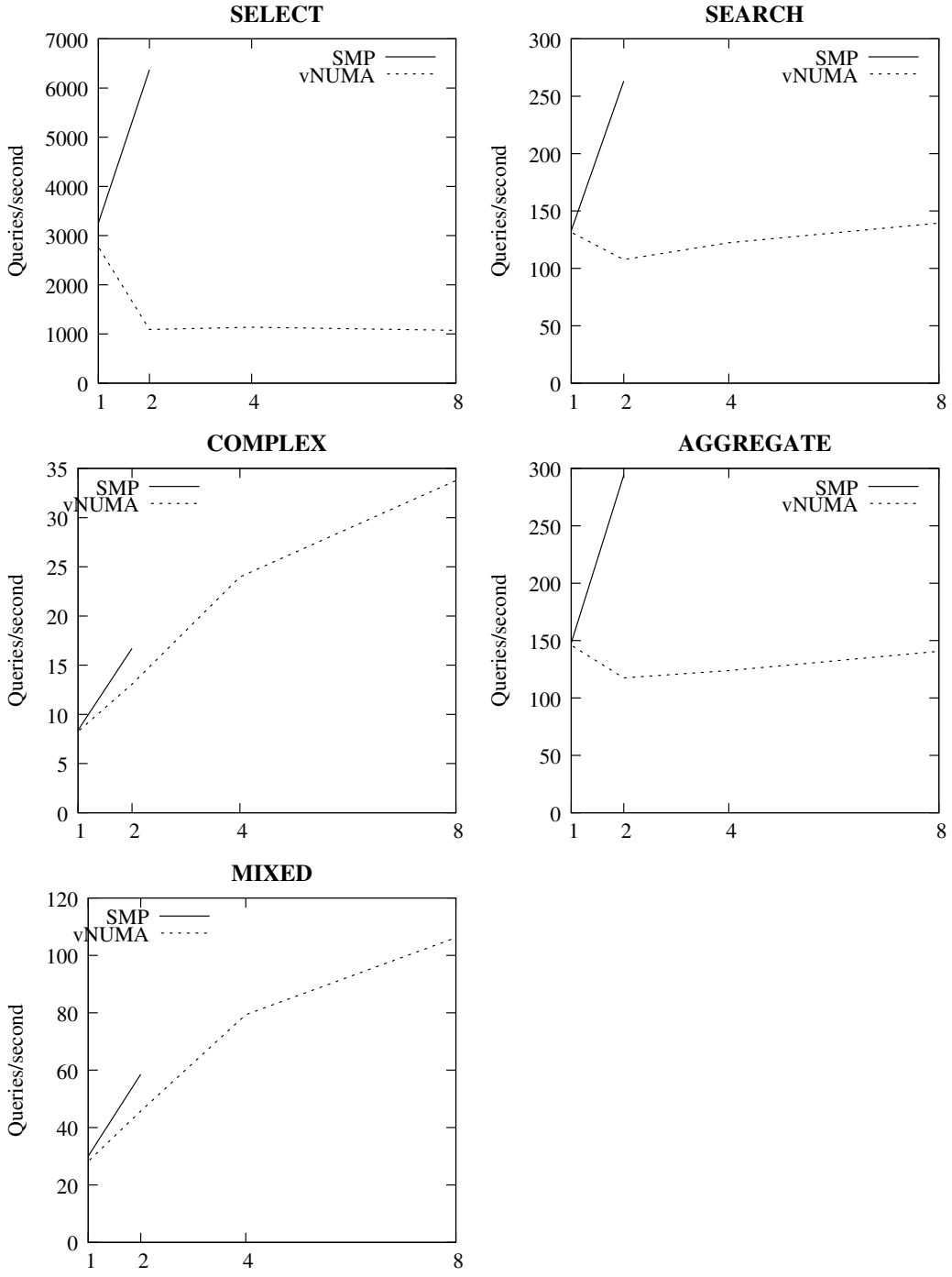



**Figure 11.4:** Database benchmark performance summary. Horizontal axes represent number of nodes.

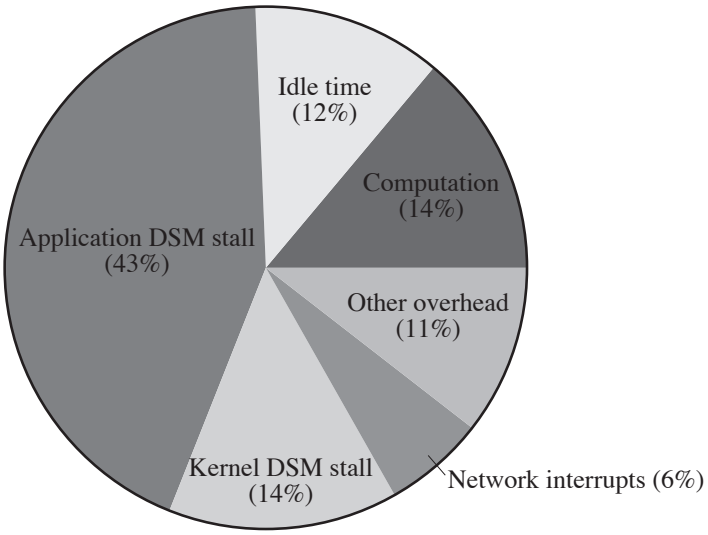



**Figure 11.5:** Time breakdown for SELECT workload

Only 14% of available processor time is used productively, which explains why the four nodes cannot match the performance of a single node. Another 12% is spent idle, which occurs when the PostgreSQL server processes are waiting to acquire locks. DSM stalls account for the largest portion of the runtime (57% of processor time), with three-quarters of those being in userspace and specifically in the PostgreSQL server processes, and the other quarter in the Linux kernel. The 11% accounted for as 'other overhead' primarily reflects the overhead of logging writes for the write-update protocol ($\approx$ 9%), with $\approx$ 2% virtualisation overhead (while **SELECT** normally experiences high virtualisation overheads, as shown in the next chapter, the fact that the it is only running 14% of the time makes the virtualisation overhead insignificant).

Since DSM stalls account for 57% of processor time usage, they deserve further attention. The vNUMA profiling tools can be used to summarise the stalls by function, as shown in Table 11.1. By far the greatest culprits are the `LWLockAcquire` and `LWLockRelease` functions, used for internal locks within PostgreSQL. Various kernel functions related to semaphores (and subsequently waking up processes) also feature highly; PostgreSQL uses semaphores to sleep when waiting on locks. `get_hash_entry` is an internal function used by the

| | |
|---|---|
| 21046 ms | `LWLockAcquire` [PostgreSQL (lightweight locks)] |
| 12156 ms | `LWLockRelease` [PostgreSQL (lightweight locks)] |
| 3439 ms | `try_to_wake_up` [kernel (semaphores)] |
| 2498 ms | `IncrBufferRefCount` [PostgreSQL] |
| 2350 ms | `get_hash_entry` [PostgreSQL (heavyweight locks)] |
| 2294 ms | `_spin_lock` [kernel] |
| 2051 ms | `update_queue` [kernel (semaphores)] |
| 1800 ms | `GrantLock` [PostgreSQL (heavyweight locks)] |
| 1547 ms | `s_lock` [PostgreSQL (spinlocks)] |
| 1426 ms | `mutex_lock` [kernel] |

**Table 11.1:** Stalls by function, **SELECT** query mix (top 10)

PostgreSQL lock manager. In other words, the major overheads are related to locking within PostgreSQL. Other functions that one might expect to feature highly in such a database workload — such as the socket communication between client and server processes — are much further down the profile and pale into insignificance.

To explain this, it is necessary to delve briefly into lock usage in PostgreSQL. There are three types of locks in PostgreSQL: spinlocks, 'lightweight' locks and regular heavyweight locks. Spinlocks are implemented using an atomic exchange operation, and have the conventional semantics. Lightweight locks are built on spinlocks, and are slightly higher-level, providing simple sharing semantics and sleeping if the lock is contended. Heavyweight locks provide even richer sharing semantics that are useful for objects such as database tables. Heavyweight locks are built on lightweight locks; however, notably, each heavyweight lock does not use its own lightweight lock, but there are a small number of contiguous lightweight locks which are used for protecting data about all of the heavyweight locks in the system[2]. Thus, contention for this small number of lightweight locks can hamper the scalability of all heavyweight locks.

A critical point to note is that, in addition to this bottleneck, the multi-layer design substantially increases the potential overheads when lock contention

[2]In fact, prior to PostgreSQL 8.2, there was only a single global lock used for this purpose. In version 8.2 this was improved such that regular locks are partitioned into 16 partitions each having a lightweight lock. However, these 16 lightweight locks still span only four cache lines on SMP, and are co-located on one page in vNUMA.

occurs. Each attempt to acquire a lock involves first locking all lower level locks. First, the underlying spinlock must be acquired (which may take multiple attempts). Once the spinlock is held, the lightweight lock can be tested; if it is found to be locked, then the spinlock is unlocked and the whole process must be tried again later. Once the lightweight lock is held, the heavyweight lock can be tested; if it is found to be locked, then the whole process has been in vain. To make matters worse, *unlocking* both lightweight and heavyweight locks also requires locking the underlying locks[3].

In a system with large and unpredictable latencies for certain operations, it is not unlikely that one of these layers will regularly encounter a lock that is temporarily locked by another node. Since each retry attempt involves locking all lower layer locks, increasing the chances of contention on those locks, this can produce an avalanche effect whereby all locks become exponentially more contended. Whenever locks are contended, communication is unavoidable, and the system becomes limited by network latency.

It is not necessarily fair to extrapolate from PostgreSQL and assume that all database software will experience such severe locking problems. Since vNUMA can provide high levels of read replication and caching — and potentially a large amount of distributed RAM that may be faster than disk — designs that allow lock-free read accesses to data, such as via read-copy-update techniques [66, 33], could theoretically provide very good performance. In this case, kernel performance would again become the ultimate challenge.

[3]Comically, this means that a failed attempt at locking a heavyweight lock requires locking a spinlock in order to unlock the lightweight lock.

# Chapter 12

# Analysis of implementation choices

The previous chapter introduced a set of benchmarks and presented sample results demonstrating the performance achievable on vNUMA. However, these performance numbers depend on many different factors and implementation choices. For the benefit of designers of future systems, it is important to be able to quantify the effects of individual design decisions. In this chapter, some of the most important decisions are discussed.

## 12.1 Pre-virtualisation

To minimise the overheads resulting from virtualisation, vNUMA uses the pre-virtualisation technique described in Chapter 4. Pre-virtualisation automatically replaces the privileged and sensitive instructions in a guest kernel with hypervisor-specific emulation code. Compared to a traditional trap-and-emulate approach — which relies on privileged and sensitive instructions trapping to the hypervisor and being transparently emulated — pre-virtualisation significantly reduces virtualisation overhead, without requiring the manual effort of para-virtualisation.

Some basic measurements that illustrate the effect of virtualisation on basic operating system code paths are listed in Table 12.1. These numbers were obtained using `lmbench` [68], a widely used suite of operating system microbenchmarks. In the table, pre-virtualisation is compared both to native execution — with no virtualisation layer — and to a trap-and-emulate virtualisation method. In the latter case, a largely unmodified Linux kernel is used, with privileged operations

| | Native | Pre-virt | Full |
|---|---|---|---|
| Fast system call[a] (`getppid`) | 0.07 | 0.09 | 0.96 |
| Slow system call (`write`) | 0.35 | 0.66 | 2.46 |
| Page fault | 1.47 | 3.13 | 8.19 |
| Context switch | 1.79 | 2.56 | 7.72 |
| Pipe communication | 7.30 | 9.90 | 30.27 |
| Socket communication (`AF_UNIX`) | 13.3 | 16.7 | 59.9 |
| Create new process (`fork`) | 133 | 218 | 503 |
| Load new program (`exec`) | 1585 | 1803 | 2848 |

[a]In the Itanium Linux kernel, some system calls use a special fast path that incurs less overhead.

**Table 12.1:** Latencies for basic operating system operations (all values in microseconds, obtained using `lmbench` 3.0). *Native* represents a kernel on bare hardware, *pre-virt* is the pre-virtualisation method, and *full* represents a trap-and-emulate form of full virtualisation in which all privileged and sensitive instructions trap to vNUMA for emulation.

still intact; a simple opcode substitution technique is used to replace problematic unprivileged instructions with opcodes that trap to vNUMA.

In each case, the virtualisation overhead is reduced significantly by pre-virtualisation. For example, a basic context switch is measured to be 1.79 $\mu$s on real hardware. This increases to 7.72 $\mu$s when using the trap-and-emulate approach, due to the high frequency of hypervisor traps; this is despite the fact that vNUMA is optimised for efficient trap handling. When pre-virtualisation is used, the context switch cost falls to 2.56 $\mu$s, which is much closer to the native value. The other measurements follow similar patterns.

While this thesis focuses on vNUMA, the author also experimented with using the same pre-virtualisation technique in the Xen/ia64 hypervisor; this similarly yielded significant performance improvements over full virtualisation and even over a kernel that had been partly para-virtualised by hand [60]. The data provided in [60] also demonstrates that, for the version of Xen/ia64 studied, vNUMA virtualisation performance surpassed unmodified Xen and was comparable to Xen with para-virtualisation retrofitted.

The practical effect of virtualisation overhead on a subset of the benchmarks studied in this thesis is shown in Figure 12.1, for both 16 KiB and 4 KiB choices of hypervisor page size[1]. Larger page sizes reduce memory management

[1]This refers to the page size used internally in the hypervisor; in both cases the guest kernel uses a 16 KiB page size.

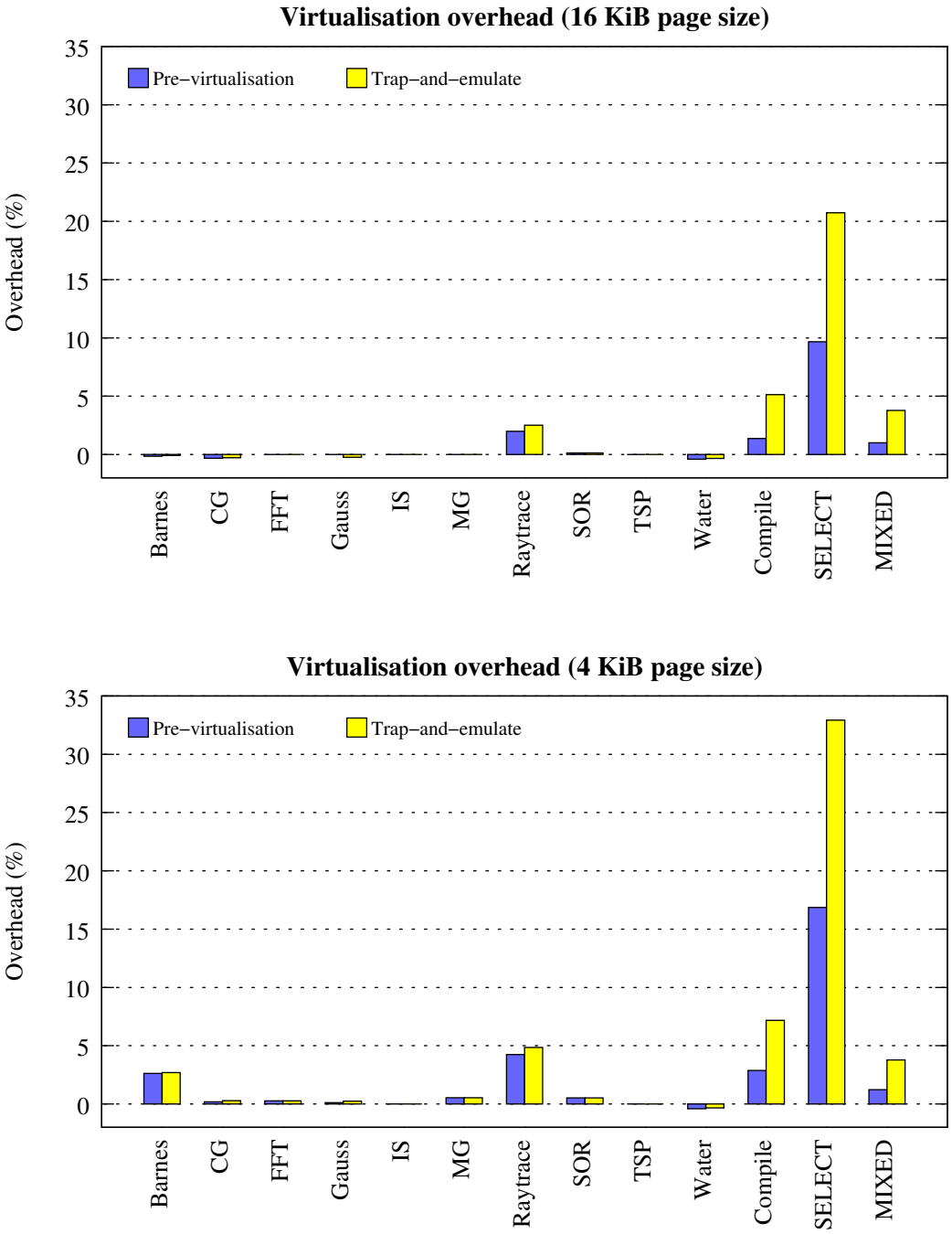



**Figure 12.1:** Virtualisation overhead

overheads, so the 16 KiB page size is a better reflection of achievable virtualisation performance. However, when the distributed shared memory system is introduced, a smaller page size becomes desirable, as will be seen in Section 12.2.3.

Not surprisingly, the greatest virtualisation overhead is incurred by the compile and database benchmarks, which involve the greatest use of operating system services. **SELECT** is the most extreme case; as previously mentioned, it is a very high throughput benchmark which executes thousands of queries per second. Each query is computationally simple but involves services such as inter-process communication and context switching, which virtualisation increases the cost of.

Pre-virtualisation is very effective in reducing overheads for these OS-intensive benchmarks; for example, in the 16 KiB case, **SELECT** overhead is reduced from over 20% to under 10%, **MIXED** overhead is reduced from 3.8% to 1.0%, and **Compile** overhead is reduced from 5.1% to under 1.4%.

In contrast, the HPC benchmarks are very much computation-dominated and show relatively low virtualisation overheads; any overheads are related to memory management. A few of the HPC benchmarks, particularly in the 16 KiB case, actually show negative virtualisation overhead: that is, they are marginally faster on vNUMA than on native hardware. This may seem unlikely, but is in fact not uncommon. Many of the HPC benchmarks have large working sets, larger than the coverage of the processor's TLB, and thus they depend on good hit rates in the next-level translation structure (the *virtual hashed page table*, or VHPT). The 'long' VHPT format used by vNUMA provides different trade-offs to the 'short' format used natively by Linux, which can affect some workloads positively and others negatively. **Raytrace** experiences a small performance degradation. (A comparison of the two VHPT formats is provided in an earlier paper [12].)

Compared to the other HPC benchmarks, **Barnes** and **Raytrace** show the greatest sensitivity to page size: for these benchmarks, the overhead of lowering the page size to 4 KiB is almost 3%. This is because they exhibit sparse memory access patterns that place the greatest demands on the processor's TLB. Since the number of entries in the TLB is fixed, using smaller pages decreases TLB coverage and therefore increases TLB miss rates.

## 12.2 DSM protocol

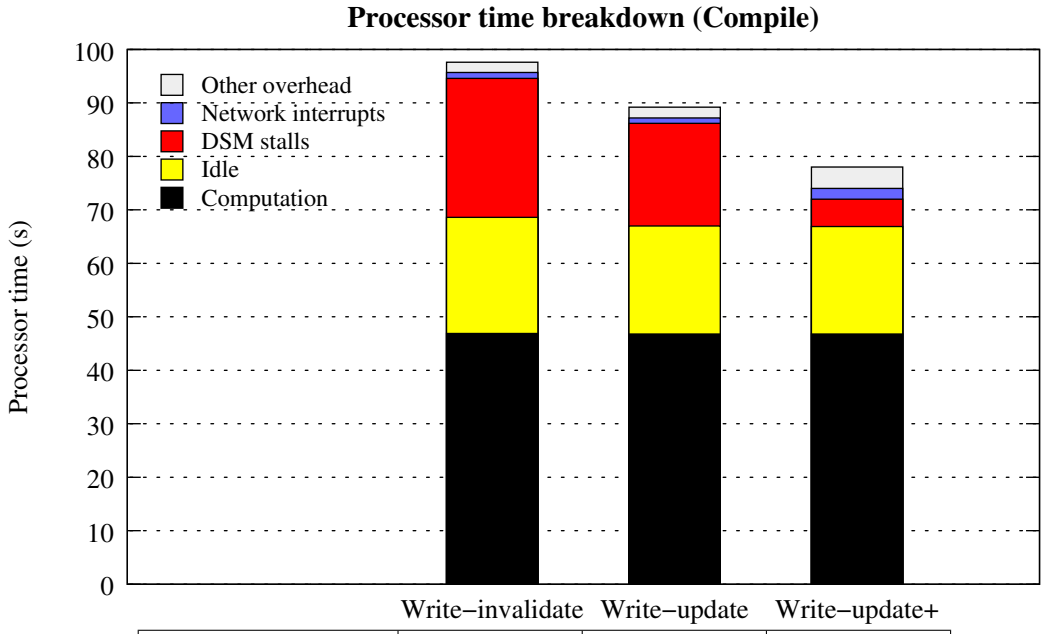
### 12.2.1 Protocol optimisations

vNUMA's distributed shared memory system also involves many design choices that should be evaluated.

The core DSM protocol is a multiple-reader/single-writer write-invalidate protocol, as described in Chapter 7. In this protocol, writing to a page always results in synchronously invalidating other nodes' copies of that page, which provides the strongest consistency guarantees but typically not the best performance. Such protocols are the most basic of DSM protocols and widely understood.

However, vNUMA also provides additional protocol optimisations to cope with particular page access patterns that result in poor performance when the core protocol is used. When the write-update protocol (described in Section 8.1) is enabled, certain pages use a multiple-writer protocol in which writes are queued and transmitted as update messages. This mitigates the impact of false sharing and enables greater caching of pages for reads. For pages where invalidation is preferable, invalidations are propagated asynchronously. The write-update+ protocol (described in Section 8.7) extends the write-update protocol to not only handle ordinary writes but also atomic memory operations, used to implement primitives such as locks and barriers. Since such operations are frequently used in conjunction with shared data structures, this can further improve performance.

Figure 12.2 summarises the performance of the compile benchmark at these different protocol levels. Performance is improved significantly by the more advanced protocols, with speed-up on four nodes increasing from 1.84 to 2.02 to 2.34. This is due to a sharp reduction in the number and latency of stalls. With the write-invalidate protocol, 420,000 synchronous stalls are required for invalidations and subsequent fetches, totalling 26.0 seconds (an average of 62 $\mu$s/stall, which is dominated by the high latency of fetching page data that is required in 66% of cases). The write-update protocol reduces the number of synchronous stalls to 285,000, with a proportional decrease in stall time to 19.2s. However, the write-update+ protocol has the most dramatic impact, reducing stall time to only 5.1s. While the total number of stalls is still 164,000, the majority



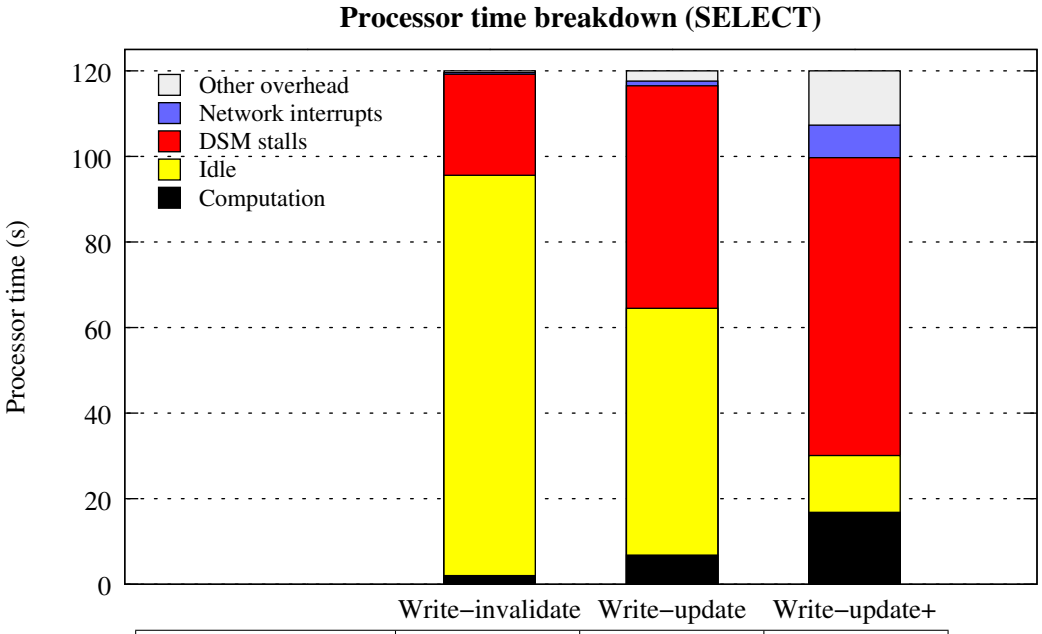

| | Write−invalidate | Write−update | Write−update+ |
|---|---|---|---|
| Total processor time | 97.6s | 89.2s | 78.0s |
| Speed-up | 1.84 | 2.02 | 2.34 |
| DSM stall time | 26.0s (27%) | 19.2s (22%) | 5.1s (7%) |
| Number of stalls | 420,000 | 285,000 | 164,000 |
| Stalls requiring data | 276,000 | 187,000 | 28,000 |
| Buffered writes | 0 | 435,000 | 3,284,000 |
| Write notices sent | 0 | 47,000 | 496,000 |

**Figure 12.2:** Compile benchmark comparison for different protocols. Stall and write counts are given to the nearest thousand.

of these are now ownership transfers, which involve minimum-length packets and therefore have low latency. The number of stalls that must fetch data has decreased to only 28,000, which shows the effectiveness of this protocol in enhancing read-caching.

The price of this improved read-caching is that many more writes must be intercepted and propagated, which is reflected in higher overheads both for intercepting the writes (reflected in hypervisor overhead) and at the receivers of the write notices (reflected in interrupt overhead). Nonetheless there is still a significant net performance improvement.

Results for the database benchmark are shown in Figure 12.3. Surprisingly, the total number of stalls actually *increases* with the more advanced protocols rather than decreasing. This can be explained by looking at the throughput and

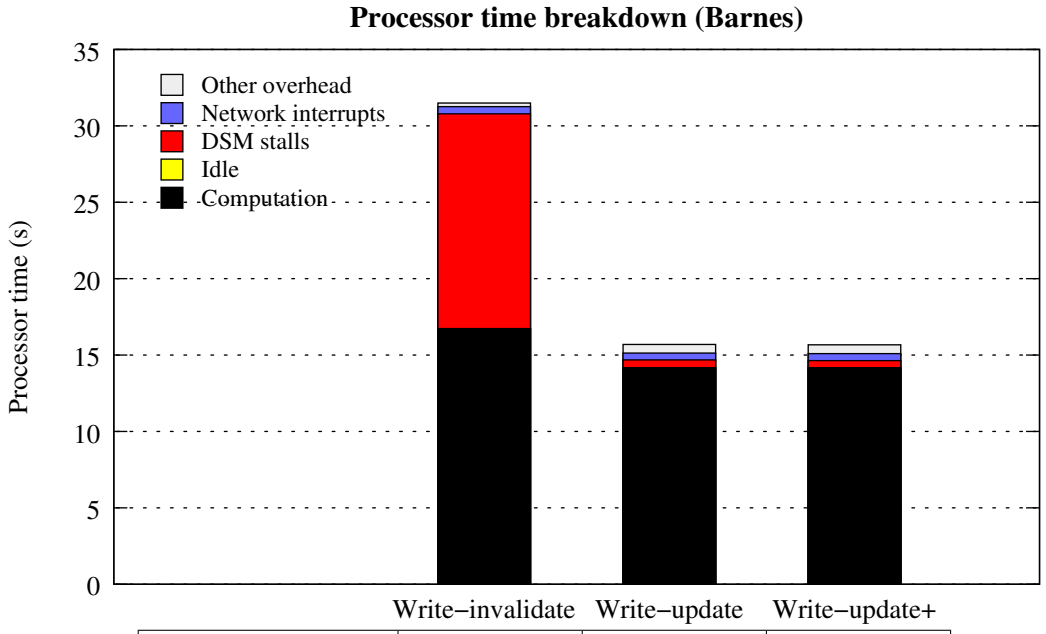



| | Write−invalidate | Write−update | Write−update+ |
|---|---|---|---|
| Total processor time | 120.0s | 120.0s | 120.0s |
| Throughput | 104 q/s | 413 q/s | 1137 q/s |
| DSM stall time | 23.6s (20%) | 52.0s (43%) | 69.6s (58%) |
| Number of stalls | 357,000 | 689,000 | 1,476,000 |
| Stalls requiring data | 231,000 | 466,000 | 80,000 |
| Buffered writes | 0 | 3,325,000 | 18,866,000 |
| Write notices sent | 0 | 293,000 | 3,522,000 |

**Figure 12.3:** Database benchmark (**SELECT**) comparison for different protocols

the idle time. With the write-invalidate protocol, PostgreSQL experiences such serious lock contention that it sleeps 78% of the time, and very little real work is achieved. This happens far less often with the write-update+ protocol; throughput is increased ten-fold and lock ownership transfers become the dominant factor limiting performance.

For the HPC benchmarks, the write-update protocols produce the most significant improvements for those benchmarks that exhibit the greatest read-write sharing (**Barnes**, **Water** and **TSP**). Performance metrics for a representative application in this set — **Barnes** — are shown in Figure 12.4, while Figure 12.5 presents a performance overview of all of the HPC benchmarks. Write-update+ has lesser effect for the HPC set of benchmarks, since most of the HPC benchmarks are less dependent on locking and synchronisation (aside from those




| | Write−invalidate | Write−update | Write−update+ |
|---|---|---|---|
| Total processor time | 31.48s | 15.68s | 15.68s |
| Speed-up | 1.60 | 3.22 | 3.22 |
| DSM stall time | 14.06s (45%) | 0.52s (3%) | 0.46s (3%) |
| Number of stalls | 245,000 | 7,000 | 6,300 |
| Stalls requiring data | 133,000 | 5,200 | 4,300 |
| Buffered writes | 0 | 1,656,000 | 1,663,000 |
| Write notices sent | 0 | 38,000 | 39,000 |

**Figure 12.4: Barnes** benchmark comparison for different protocols

that are barrier-intensive, such as **Gauss**). **SOR** is the one benchmark that gains no benefit from write-update or write-update+; it is specifically designed to have no false sharing, and the write-invalidate method of propagating data is essentially optimal for it.

The fact that the write-update+ protocol has such dramatic effects on the compile and database workloads bears noting, as it has important implications for future vNUMA-like systems. The only difference between the write-update and write-update+ protocols is the treatment of atomic operations, yet this has significant effects on the number of writes buffered, the number of remote reads, and overall performance. The reason is that many shared data structures, both in the Linux kernel and in applications designed for SMP systems, have embedded locks and/or reference counts. Without the enhancements in the write-update+

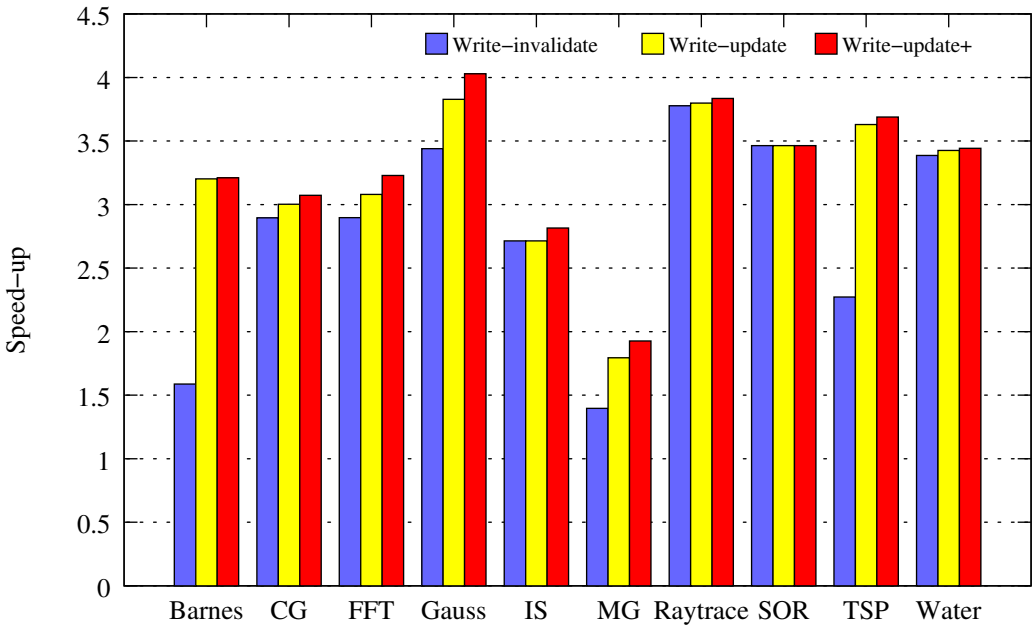



**Figure 12.5:** HPC benchmark performance for different protocols

protocol, pages with such data structures always fall back to write-invalidate mode, and there is little benefit from advanced protocols. Yet it is precisely such data structures — frequently accessed by multiple nodes and often with false sharing problems — that can stand to benefit most from novel schemes that allow simultaneous readers and writers.

### 12.2.2 Invalidation threshold

With the write-update protocols, it is rarely *necessary* to invalidate a page. Ordinary write operations can always be buffered and propagated as updates, and the write-update+ protocol even allows similar treatment for atomic memory operations. Thus, most pages can become widely cached for reading, with the write-update protocol used to propagate updates. An invalidation is only forced in certain limited cases — for register stack engine accesses, for certain infrequent Itanium instructions that are not handled in vNUMA, and when the Linux kernel explicitly requests it (in the para-virtualised `clear_page` function).

As proven by the results in the last section, this increased caching can provide

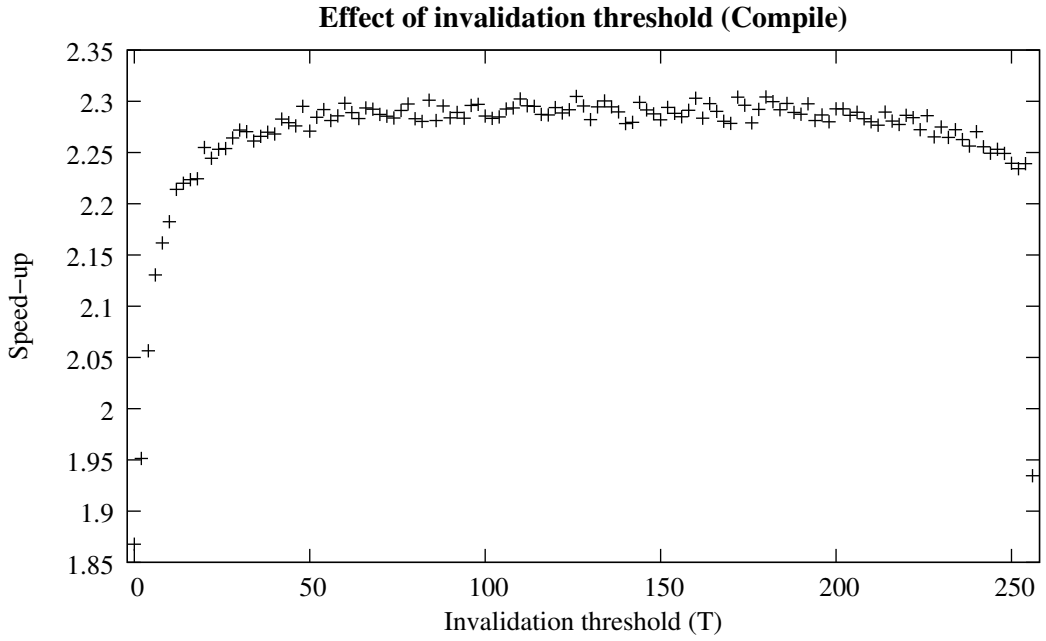



**Figure 12.6: Compile** performance at different invalidation thresholds (T)

significant benefits by reducing the frequency with which read instructions must stall. However, if the write-update protocol were to be used for every page in the system, the number of buffered writes and write notices sent could increase by many orders of magnitude (for example, the total number of writes during a compile benchmark run is around $2 \times 10^9$). The resulting increase in overheads would negate any benefit, and indeed have severe negative effects. A critical part of the write-update protocols, therefore, is determining when *not* to use the write-update protocols — in other words, when to invalidate a page instead.

In vNUMA, the scheme described in Section 8.4 is used. Briefly, each page starts off in write-update mode, and vNUMA tracks the number of successive local writes with no intervening remote accesses. Once this 'run' of local writes reaches a certain threshold, the page is invalidated — it is assumed that either the node is using the page exclusively, or that the node is making a large set of changes that would be inefficient to propagate individually. This scheme has the advantage of being simple and minimal in overhead. However, it does depend on tuning of the threshold parameter.

Figure 12.6 shows compile performance for different invalidation thresholds

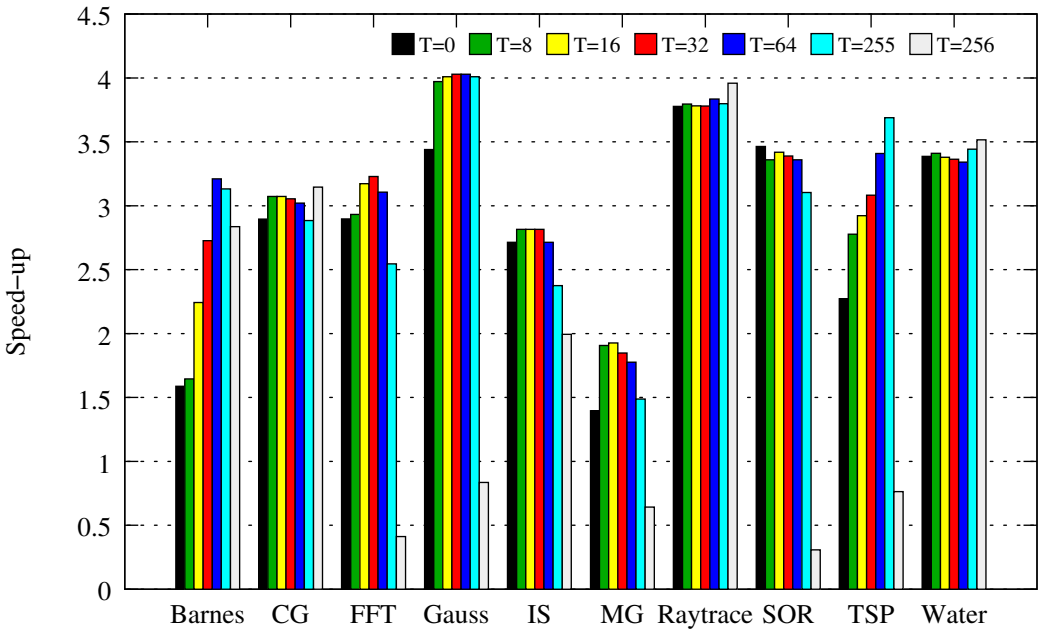



**Figure 12.7:** HPC benchmark performance at different invalidation thresholds (T)

$T$. Since vNUMA's per-page write counter is an 8-bit number and never reaches 256, $T = 256$ is equivalent to $T = \infty$: in other words, a pure write-update protocol. $T = 0$ represents a pure write-invalidate protocol. The graph shows that both of these extremes are non-optimal, and the optimal performance is achieved for a hybrid adaptive protocol with $T$ between around 50 and 200. There is low sensitivity to the exact value of this parameter.

The basic relationship holds for most workloads, with varying degrees of curvature depending on the amount of false sharing (which penalises a pure write-invalidate solution) and number of writes (which penalises a pure write-update solution). For certain simple benchmarks, write-invalidate can be optimal, in which case performance degrades slightly for increasing values of $T$ (this represents unnecessary write-update overhead before each page falls back to write-invalidate mode). This is the case for **SOR**. Some benchmarks also happen to perform well with a pure write-update scheme; however in the majority of cases this results in severe performance degradation due to the volume of writes propagated. The effect of varying the threshold on the various HPC benchmarks is summarised in Figure 12.7.

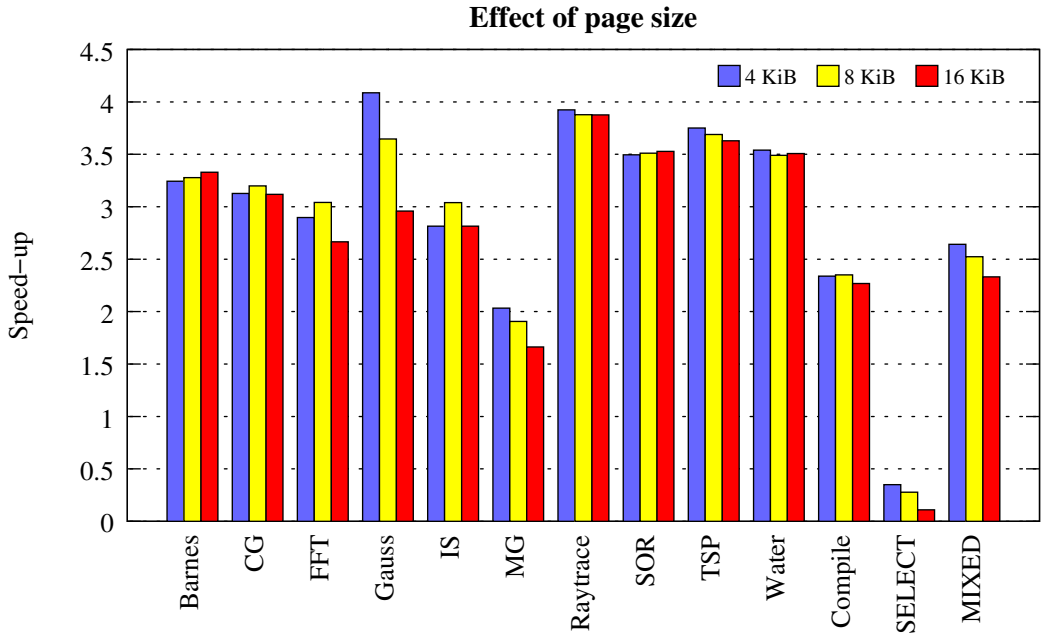



**Figure 12.8:** Benchmark performance at different DSM page sizes

### 12.2.3 Page size

The Itanium architecture does not have a fixed or preferred page size; the page size can be chosen out of a large set of options. For a DSM system, the most obvious choice is to use the minimum supported page size (4 KiB). This is because one of the big challenges facing DSM systems is false sharing — unrelated data co-habiting the same page and causing undesired contention — and this is minimised by using the smallest page size.

Nonetheless, there are some disadvantages of using such a small page size. Firstly, it increases memory management overheads in the hypervisor, as was shown in Section 12.1. Additionally, some DSM workloads — those that have little false sharing and access memory relatively contiguously — can benefit from a larger page size, since more data is transferred per fault.

Given that the vNUMA DSM protocol can cope reasonably well with workloads that involve false sharing (such as **Barnes**, **Water** and the Linux kernel itself), it is interesting to investigate the effect of changing the page size. Results on four nodes for various page sizes are shown in Figure 12.8.

**Barnes** and **SOR** benefit from larger page sizes even up to 16 KiB. For **SOR**, this is because it's fundamental unit of sharing is 16 KB: contiguous blocks of 16,000 bytes are transferred in each step of the benchmark, so a 16 KiB page size is ideal. In the case of **Barnes**, it is due to its high memory management overheads at smaller page sizes, combined with the fact that the write-update protocol is effective in addressing its false sharing; thus an overall gain is achieved from the larger page sizes.

In all other cases, 16 KiB pages degrade performance; thus 16 KiB does not seem to be a good general-purpose default for DSM page size. On the other hand, 8 KiB produces good performance for a larger number of the benchmarks. In fact, even **Compile** — the DSM performance of which is limited by the Linux kernel — benefits very slightly from 8 KiB pages; again this is due to the reduced memory management overhead. Unfortunately, 8 KiB pages produce significant degradation for **Gauss**. In **Gauss**, the unit of sharing (a matrix row) is 4 KiB, so performance degradation occurs at higher page sizes, both due to unnecessary data transfer and due to false sharing.

Ultimately the choice of page size is very much workload-dependent. 4 KiB pages are used throughout this thesis in order to minimise the number of cases in which false sharing arises. However, workloads which have large working sets and transfer data in large blocks can benefit from larger DSM page sizes. One of the advantages of a hypervisor-based DSM system is that the DSM page size can be varied independently of the operating system page size.

### 12.2.4 Coherence algorithm

Another aspect that warrants evaluation is the overhead of the coherence algorithm described in Section 8.6. In the benchmarks thus far, this algorithm was not enabled, since all of the tested workloads execute correctly without it. Nonetheless it may be useful when stronger coherence guarantees are required.

Microbenchmarks show that, using the data structures described in this thesis, the coherence algorithm increases the cost of intercepting a write from 243 cycles to 268 cycles, and the cost of applying a remote write from 56 to 91 cycles (in the optimal case that the chains do not need to be traversed, i.e. there are no

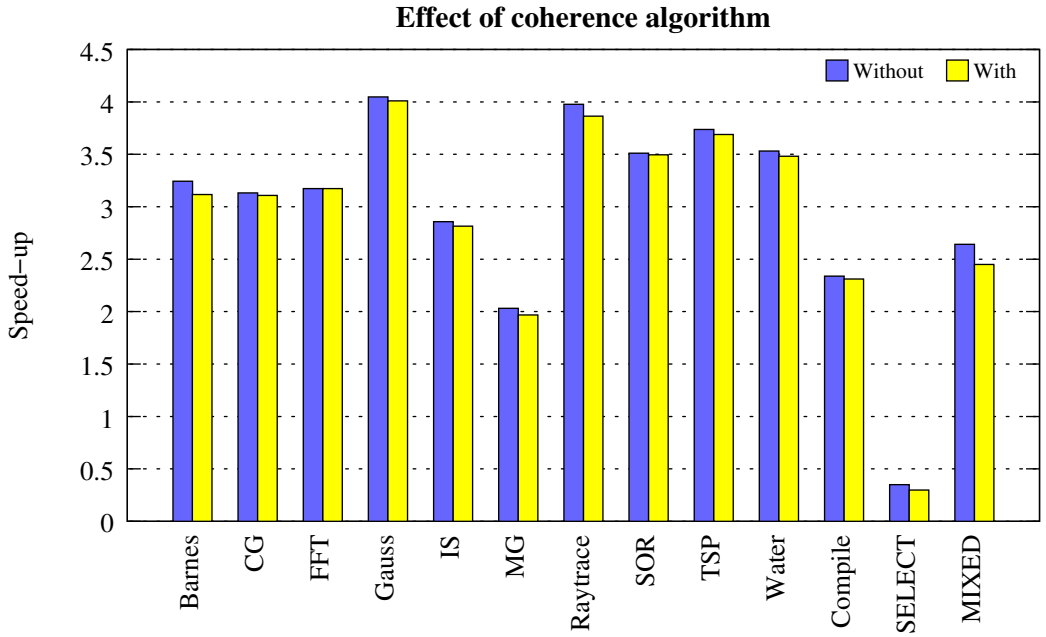



**Figure 12.9:** Performance effects of enabling coherence algorithm

hash table conflicts and no newer writes that conflict). These overheads seem quite reasonable. As always, they could be reduced by implementing the write interception code in assembly language rather than C, but at the expense of maintainability.

The performance of the various benchmarks with the coherence algorithm enabled is presented in Figure 12.9. Not surprisingly, the overhead is greatest for those benchmarks which generate the most write-update traffic, namely **Barnes** and the database benchmarks. It is also quite high for **Raytrace**, despite the fact that it does not generate as many writes as some of the other benchmarks; this seems to be because this benchmark stresses the processor caches (due to its sparse access pattern, like **Barnes**) and so the extra cache footprint of the coherence tracking structures has a greater effect. However, overall, the overheads are acceptable.

### 12.2.5 Release detection

One interesting feature of the vNUMA DSM is the use of release consistency annotations — a feature of the Itanium architecture — to optimise the generation of write notices. Store instructions in the Itanium architecture provide no ordering guarantees, unless they are of the *release* form, which indicates that they must become visible after any previous operations by the same processor.

vNUMA utilises this release consistency mechanism as described in Section 8.8. Consider when a vNUMA node receives a fetch request for a page that is exclusively owned and may therefore contain local writes. At the same time, vNUMA may be buffering writes to other pages. Since it does not track every write to the exclusive mode page, it does not know how the two sets of writes were interleaved in the source program; to guarantee processor consistency, it would have to ensure that they are made visible simultaneously (by flushing buffered writes before the fetch reply). This slightly increases network traffic and the latency of the fetch reply. However, given release consistency, this flush can be avoided in many cases; unless a release-type store has been made to the exclusive page, it is safe to allow the fetch to bypass the buffered writes.

Experiments show that the effect of this optimisation is very minor; benchmark results are not affected to any degree of statistical significance. This is primarily due to the fact that write buffer flushes are sufficiently common that the occasional avoided flush is lost in the noise. An additional factor is that, even with the optimisation enabled, the release detection check often returns a false positive. This is because vNUMA can only determine (using the Itanium performance monitoring unit) whether a release-type store has been made in the last interval, and not to what specific page it was made.

This result is a mixed blessing. While it is a pity that the effect of this novel technique is so insignificant, it is also a technique that is only feasible on the Itanium architecture. The fact that the vNUMA DSM performs well without it bodes well for implementations on other architectures.

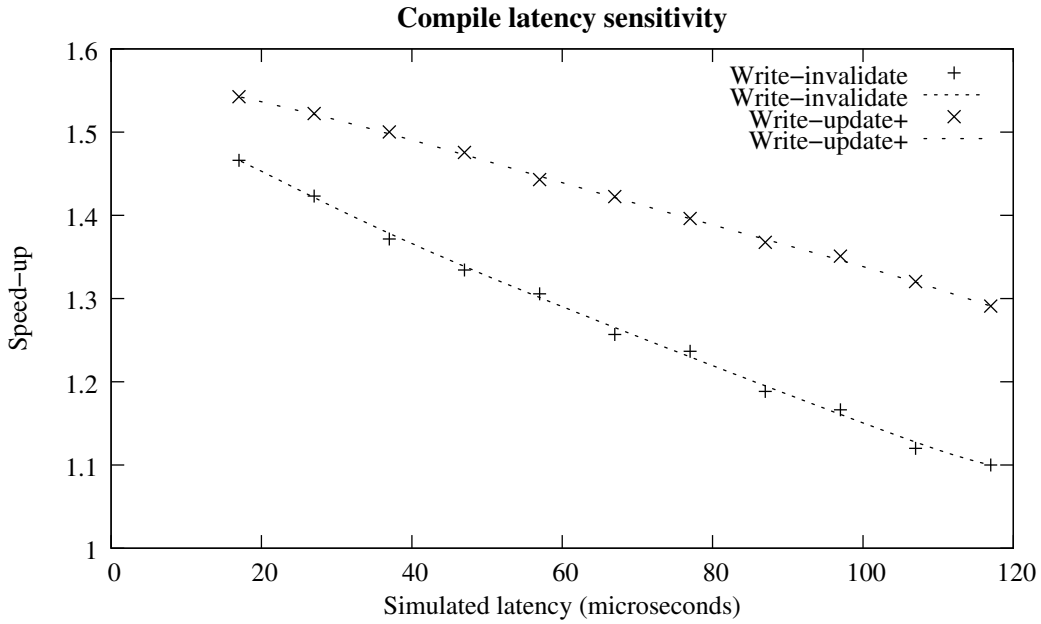



**Figure 12.10:** Compile performance for different simulated latencies (2 nodes)

### 12.2.6 Latency sensitivity

Throughout this thesis, the importance of latency has been stressed. In the vNUMA DSM system, like most other DSM systems, there are certain events which cause execution to be stalled; when such an event is encountered, it is not possible to continue the execution of the program until the appropriate data is fetched or synchronisation is performed. The program stalls for a period of time equal to the communication latency. In vNUMA, this time is wasted. In some multi-threaded DSM systems, it is possible to use some of this time usefully by switching to another thread, and this would be a worthwhile future improvement for vNUMA. This is no magic bullet, however, and minimising communication latency is still important.

Different workloads will have different sensitivity to latency. Primarily, the sensitivity to latency depends on the frequency of DSM stalls[2]. Figure 12.10 shows an example plot of latency sensitivity for the compile benchmark. (This graph was produced by artificially introducing extra latency into the communica-

[2]Of course, even if stalls could be eliminated completely, there are still cases in which network latency is important; for instance, if a receiver is waiting on an update from another node.

tion path.) While vNUMA's write-update+ helps reduce the sensitivity to latency by reducing the number of stalls, latency is still a very important parameter. A network card driver that is not optimised for latency can easily add 100 $\mu$s or more, resulting in performance that is at the right of the graph rather than the left.

Given the importance of latency for many applications, it is surprising that latency is so often neglected in both hardware and software design. Compared to Linux's UDP round-trip latency of 63 $\mu$s, vNUMA achieves 17 $\mu$s round-trip latency on the benchmarked systems. As demonstrated by the graph, this in itself can provide significant performance advantages.

Note that the quoted 17 $\mu$s latency includes the HP Procurve switch; with two nodes connected back-to-back, this round-trip latency can be reduced to 12 $\mu$s. This is still disappointingly high, however, when one considers that a minimum length packet only takes 576 ns to be transmitted on the Ethernet. Nevertheless it seems to be close to the optimal performance achievable on today's commodity systems; the GAMMA project, which seeks to provide low-latency messaging for Linux clusters, reports similar results [14].

The latency of 6 $\mu$s each way can be explained as follows. In a traditional network card architecture, each packet send requires writing to a device register to trigger the send (where the CPU-to-device latency is typically of the order of 500ns). Then, the device must perform at least two DMA reads, one for the packet descriptor and one for the packet data (each DMA read incurs a minimum of 600–700ns latency in the system under test, as measured with a PCI bus analyser). Inevitably there is also some internal latency within the network device before and after these reads, which can be of the order of a microsecond. Finally, the packet is transmitted onto the Ethernet (576 ns). The receiving card incurs some internal latency, performs one DMA write with the packet data, performs another DMA write to update the in-memory receive descriptor, and toggles its interrupt line. Some 700ns later, the destination CPU receives an interrupt. These basic latencies can easily add up to 6 $\mu$s or more.

There are some important considerations when optimising a network driver for latency. Since each DMA read involves significant latency to set up, the number of packet fragments should be minimised, and copying data into a single buffer may be preferable unless CPU time is at a premium. To ensure that receive interrupts

are delivered as early as possible, any interrupt throttling features of the network card should generally be switched off. Finally, software overhead should not be underestimated; small optimisations such as filling in packet headers ahead of time can make a difference. If possible, the device should be configured so that a send can be triggered with a single write and there are no device reads on the criticial path.

### 12.2.7 Transferring page data

As well as short synchronisation packets, vNUMA must also frequently transfer large packets, containing an entire page of data (4 KiB). Minimising the latency of these packets requires special consideration.

The standard Ethernet frame size limit (1518 bytes excluding preamble) implies that it is necessary to fragment a page into at least three packets. However, many Ethernet devices support 'jumbo' frames — frames larger than the normal maximum, typically up to around 9 KiB — in which case it is theoretically possible to transmit a page in a single packet.

The obvious choice would be to transmit the page data in as few packets as possible, since each extra packet adds some overhead. However, using multiple smaller packets can be advantageous for latency, by enabling pipelining. This is illustrated graphically in Figure 12.11 and 12.12.

In the case where the entire page data is transmitted in one packet, each stage of the communication is serialised (Figure 12.11). Consider the reply phase of the fetch, depicted in lighter shading. First, the entire packet data must be transferred over the PCI bus before Ethernet transmission can start[3]. In the figure, this is labelled P1 PCI. For a vNUMA packet containing page data (one transmit descriptor pointing to 32 bytes of headers, and one transmit descriptor pointing to 4096 bytes of data), this stage takes around around 13 $\mu$s. Then, transmitting the page over the first Ethernet segment (Net1) takes 33 $\mu$s. Given a store-and-forward switch design — which was the case for both of the switches available for testing — the switch waits to receive the entire packet. The switch inspects the packet,

[3]Because an Ethernet frame must be transmitted contiguously and PCI does not provide any timing guarantees for supplying data, it is not safe to start transmission before the entire packet is buffered on the network card.

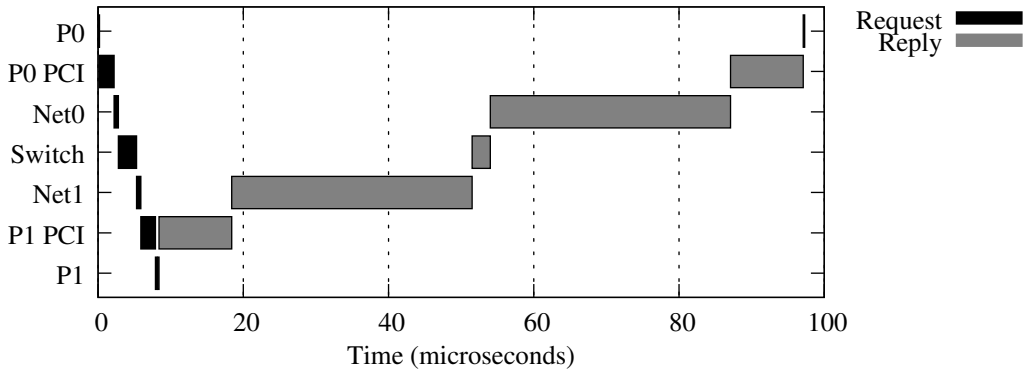



**Figure 12.11:** Using a single jumbo frame, page fetch latency is $\approx 97\mu$s

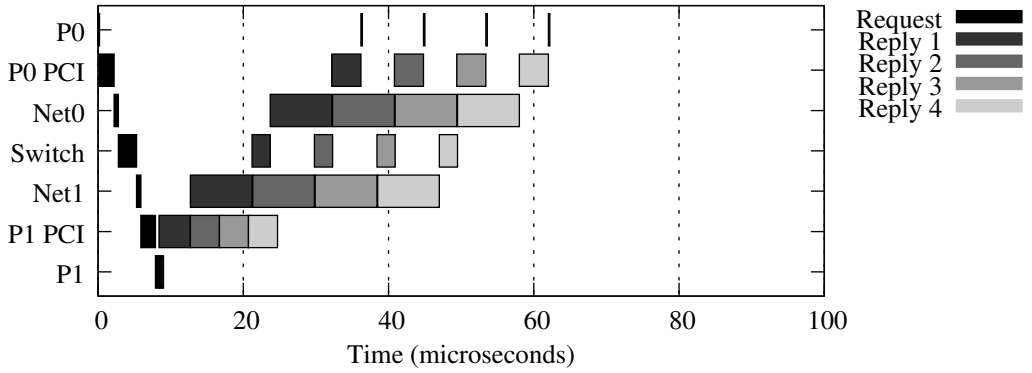



**Figure 12.12:** Splitting the page into four fragments reduces latency to $\approx 62\mu$s

routes it to the appropriate output port (around 2.5 $\mu$s for the tested switches), and begins transmission on the second network segment (Net0, which takes another 33 $\mu$s). Finally the packet is received at the destination node and transferred over PCI to memory (P0 PCI)[4] Overall, this takes around 89 $\mu$s, producing a 97 $\mu$s turnaround time for the page fetch.

In contrast, consider if the page data is transmitted in four fragments of 1024 bytes each (Figure 12.12). For each fragment, the PCI transfer takes around 5$\mu$s, and transmitting the fragment on the Ethernet takes 8.5 $\mu$s; repeating the same calculations as before, this means that the first fragment arrives after around 26.5 $\mu$s. The remaining fragments can be pipelined through the stages; for instance,

[4]Some parts of this DMA can occur in parallel with reception if the network card is configured to do so.

the second fragment can commence PCI DMA while the first fragment is being transmitted on the Ethernet. The overall latency in this case is 54 $\mu$s, allowing the page fetch to return data in 62 $\mu$s — a significant improvement over 97 $\mu$s.

Increasing the number of fragments can reduce latency even further, however it also adds to overheads. Both PCI and network bandwidth demands become greater because of the extra packet descriptors and packet headers that must be transferred. Also, the receiver incurs the CPU overhead of an interrupt for each packet, which can be significant if a large number of fragments are used.

vNUMA can be configured to use four different options: a single jumbo packet, three packets (of 1408, 1408 and 1280 bytes), four packets (of 1024 bytes each), and eight packets (of 512 bytes each). The respective best-case fetch latencies are around 97 $\mu$s, 68 $\mu$s, 62 $\mu$s and 52 $\mu$s, as predicted by the pipelining model and verified experimentally.

Experiments show that the four packet configuration is optimal, performing better than the other options; therefore it was used for the reported results. The eight packet configuration results in higher overheads and does not scale as well.

### Interactions

Thus far, round-trip latencies have been quoted assuming a quiescent system. However, plotting a histogram of stall latencies in a real application reveals an interesting anomaly. A number of applications produce distributions such as that shown in Figure 12.13 (this was produced from a **FFT** benchmark run on 2 nodes). The leftmost peak, at around 17 $\mu$s, represents short requests that do not return page data. The center peak, at around 62 $\mu$s, represents the latency of fetching page data (at a fragment size of 1024 bytes). What, then, is the rightmost peak in the graph, at around 112 $\mu$s?

This right peak corresponds to a case where a fetch request is delayed behind a fetch reply to another node, as shown in Figure 12.14. The large latency of the fetch reply is added to the normally small latency of the fetch request. The chance of such non-optimal behaviour grows with increasing numbers of nodes, since it becomes increasingly more likely that a request is delayed behind outgoing page data for another node.

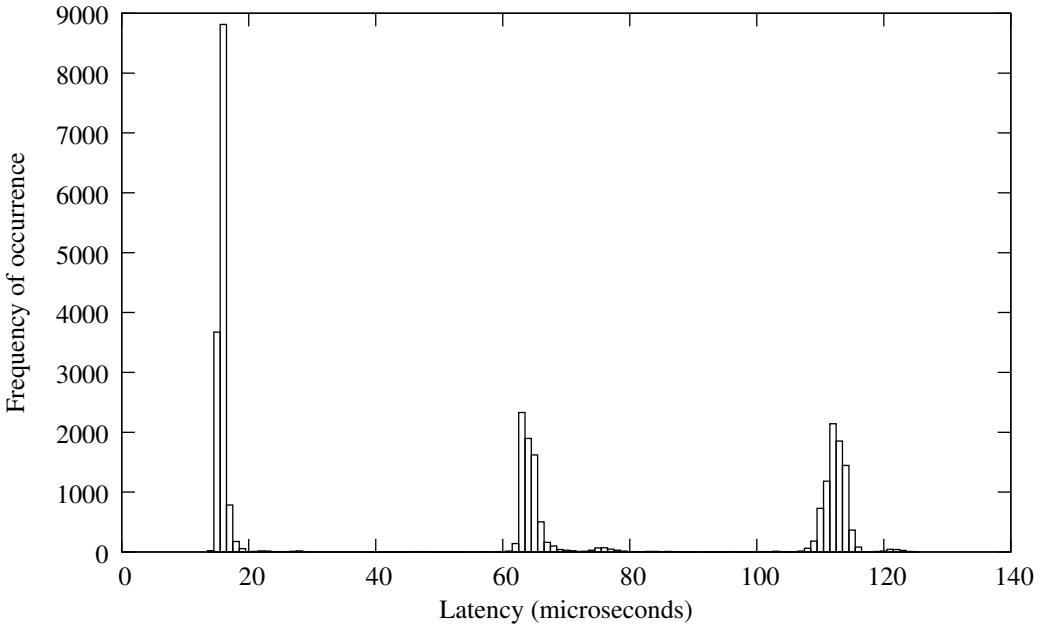



**Figure 12.13:** Stall latency histogram for **FFT** on 2 nodes

Assuming that page fragmentation is being used, it is possible to improve performance for such cases by allowing outgoing requests (which are typically the result of stalls, and therefore latency-critical) to be transmitted ahead of pending page fragments. This is shown in Figure 12.15. The latency saving for the short packet (up to 26 $\mu$s) is much greater than the extra latency incurred by the page fragments (less than a microsecond), so this is invariably a good optimisation.

This improvement can easily be implemented by maintaining two transmit queues: a high priority queue and a low priority queue. The network adapters used for this experimental work (based on the DP83820 chipset) support multiple priority queues by design, so the implementation is trivial. For network cards that do not have this support, priority queuing can be emulated in software by queueing low priority frames only when the network card becomes idle. The priority queueing technique provided slight performance improvements for some benchmarks, with no negative side-effects, and was therefore enabled for the results reported in other sections.

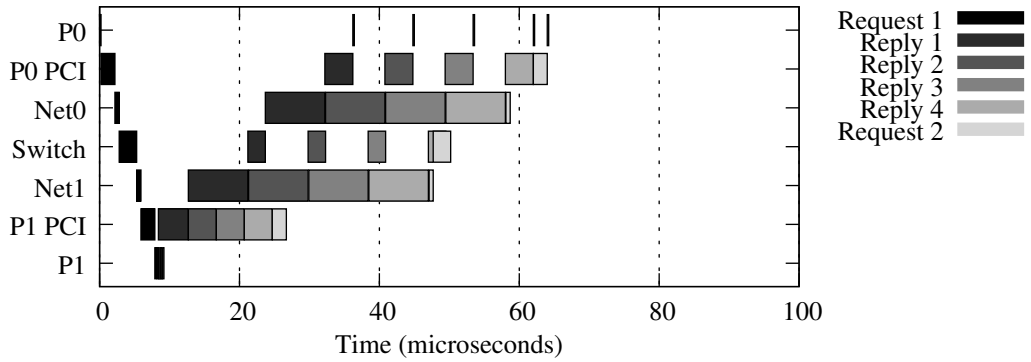



**Figure 12.14:** Request 2 incurs large latency as a result of being delayed behind the page fragments

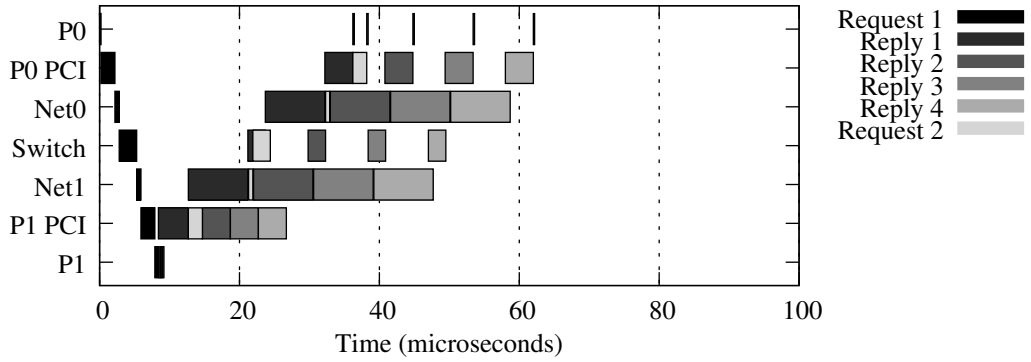



**Figure 12.15:** Using priority queueing to reduce Request 2 latency

# Chapter 13

# Conclusion

This thesis has explored the challenges associated with building a hypervisor-based multiprocessor. The result, vNUMA, is impressively usable for a variety of applications. It uses several novel techniques which have been described in this thesis, both to minimise virtualisation overhead and to optimise the performance of the distributed shared memory.

For computationally dominated applications, the system performs well, and there are compelling arguments for using such a system on a small compute cluster. The user gains the benefit of a single system image, indistinguishable from a single computer; there is a single filesystem, a single process space, and so on. The operating system performs automatic migration and load-balancing of processes between processors, just as it would on a SMP or ccNUMA system. If finer control is desired, processes and even entire login sessions can be locked to particular nodes, using the standard operating system mechanisms. All of this incurs little overhead for a CPU-bound application. Pages that are used locally by a process become owned by the node that process is executing on, and incur no extra overheads. When pages are shared between multiple processes that execute on different nodes, the user effortlessly gains the services of a high-performance DSM implementation.

Merely by virtue of being lower in the software stack, the vNUMA DSM can achieve significant performance advantages over a userspace DSM such as Treadmarks. It can incorporate its own network drivers, optimised to minimise latency for the particular packets it sends; it can bypass the inefficiencies of OS

mechanisms such as `mmap` and signals; it can intercept individual write faults with reasonably low overhead; and it can directly harness hardware resources such as Itanium's performance monitoring unit in order to make better decisions. Such optimisations would also available to a DSM implemented at the operating system level, but the vNUMA approach has the advantage of not requiring intrusive modifications to the operating system. This makes it more practical both for legacy operating systems and also rapidly evolving operating systems such as Linux.

The disadvantages of implementing the DSM at such a low layer — primarily, the fact that less application-specific information is available — can be mitigated by adding a minimal hypervisor interface to allow hints to be provided about certain pages. In vNUMA, such a technique is used to invalidate a page when the Linux kernel intends to clear it, avoiding the need to acquire ownership of the old data.

vNUMA's sophisticated DSM protocol can also competently handle workloads with sparse memory accesses, false sharing and atomic read-modify-write operations — such as the Linux kernel itself. This is achieved via an adaptive protocol that can operate in both write-invalidate and write-update modes. As demonstrated in this thesis, the write-update mode can provide significant advantages, allowing the maintenance of high levels of read replication even in the presence of writes, and thereby reducing the frequency with which the DSM system must stall waiting on page data. A particularly novel scheme is used for emulating atomic read-modify-write operations to a page, which can again avoid the necessity for invalidation of data. Naturally, replicating all data cluster-wide would be prohibitive, so vNUMA uses an adaptive algorithm that decides whether to update or invalidate on a per-page basis.

With these advanced protocols, vNUMA performs competitively even for complex workloads such as a software build. On a small cluster vNUMA can transparently achieve compile performance comparable to `distcc` — a custom software solution for distributing portions of a compile to cluster nodes — while appearing like a regular SMP system to users.

Of course, vNUMA cannot magically implement synchronisation without communication, and performance does suffer for certain workloads that make

significant use of synchronisation primitives (specifically atomic memory operations). By their very nature, atomic operations to a memory location must be ordered, and if these operations occur on multiple nodes, then communication latencies cannot be avoided. While vNUMA attempts to do this as efficiently as possible, its 17 $\mu$s latency for acquiring ownership of a page is still two orders of magnitude more expensive than a cache line transfer in an SMP system. Workloads which depend on being able to achieve a high throughput of such synchronisation operations, such as the PostgreSQL benchmark studied in this thesis, may therefore become performance-limited by this latency. In most such cases performance could be improved by re-designing the application, but one of the major motivations of the vNUMA architecture is transparent support for legacy applications.

This 17 $\mu$s communication latency that vNUMA incurs is largely dominated by the latency of the I/O architecture between the CPU and network device, particularly the DMA latency, and only secondarily determined by the network itself. For example, a minimum-length frame can be transferred over Gigabit Ethernet in a little under 600 ns, but the end-to-end latency in the system under test is at least 6 $\mu$s; similar latencies are reported by other researchers [14]. Ironically, even when vNUMA is run across cluster nodes that are physically separated in different areas of a building, the intra-node legs of the journey still dominate.

One approach that could be used to mask this latency is to use multiple threads of execution, which at the hypervisor layer would correspond to simulating multiple virtual processors per physical processor. When one thread is waiting on remote communication, switching to a different thread could potentially allow useful work to be done.

Hopefully emerging hardware developments will also enable lower communication latencies. Higher bandwidth demands and ever-improving fabrication technology are resulting in greater integration of memory and I/O interfaces directly into processors. Sun's recent UltraSPARC T2 processor [30] even integrates dual low-latency 10 Gigabit Ethernet interfaces on the processor die. Such architectures open up exciting possibilities for a system like vNUMA. Given low enough communication latencies, a network of such processors could potentially be used as a practical substitute for an SMP system, without the

engineering complexity of implementing SMP in hardware.

Simultaneously, increasing processor speeds and parallelism are also increasing the demands on the memory hierarchy, even in hardware-based SMP systems. Ultimately, it may be that current SMP memory models — based on notions of strict global coherence and atomicity — will reach their scalability limits, and operating systems and applications will need to be able to support weaker consistency models. Such developments could offer new opportunities for a hypervisor-based software DSM.

Implementing a DSM system inside a hypervisor opens up a large space of previously unexplored possibilities. This thesis has attempted to explore as much as practical of the problem space, yet there are still many possibilities that are uncharted. The author hopes that this thesis provides valuable insights that can be used as a basis for future research and development.

# Bibliography


[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2006.

[2] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual*, August 2007. http://developer.amd.com/devguides.jsp.

[3] Mustaque Ahamad, Ranjit John, Prince Kohli, and Gil Neiger. Causal memory meets the consistency and performance needs of distributed applications. In *Proceedings of the 6th SIGOPS European Workshop*, pages 45–50, 1994.

[4] J. K. Archibald. A cache coherence approach for large multiprocessor systems. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 337–345, 1988.

[5] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[6] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for Linux. In *Proceedings of Linux Expo '99*, pages 95–100, 1999.

[7] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.

[8] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15:412–447, 1997.

[9] John B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29:219–227, 1995.

[10] John B. Carter, John K. Bennett, and Will Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on OS Principles*, pages 152–64, 1991.

[11] Matthew Chapman, Daniel J. Magenheimer, and Parthasarathy Ranganathan. MagiXen: Combining binary translation and virtualization. Technical Report HPL-2007-77, Hewlett-Packard Laboratories, 2007.

[12] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, May 2003.

[13] Peter Chubb, Matthew Chapman, and Myrto Zehnder. [para]virtualisation without pain. In *Proceedings of the 8th Linux.Conf.Au*, January 2007.

[14] Giuseppe Ciaccio. GAMMA: The Genoa Active Message MAchine: Performance. http://www.disi.unige.it/project/gamma/, accessed on 30th September 2008.

[15] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 273–286, 2005.

[16] Hewlett-Packard Development Company. SKI IA-64 simulator. URL http://ski.sourceforge.net/.

[17] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal for Research and Development*, 25(5):483–490, September 1981.

[18] R.C. Daley and J.B. Dennis. Virtual memory, processes, and sharing in Multics. *Communications of the ACM*, 11(5):306–312, May 1968.

[19] Partha Dasgupta, Richard J. LeBlanc, Jr, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, 1991.

[20] Alba Cristina Magalhães Alves de Melo. Defining uniform and hybrid memory consistency models on a unified framework. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, volume 8, pages 270–279, 1999.

[21] Xavier Defago, Andre Schiper, and Peter Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[22] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[23] Dolphin Interconnect Solutions. SCI benchmarks. http://ww.dolphinics.no/products/benchmarks.html, accessed on 30th September 2008.

[24] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. *SIGARCH Computer Architecture News*, 14(2):434–442, 1986.

[25] Hideki Eiraku and Yasushi Shinjo. Running BSD kernels as user processes by partial emulation and rewriting of machine instructions. In *USENIX BSDCon '03*, pages 91–102, 2003.

[26] M. Rasit Eskicioglu, T. Anthony Marsland, Weiwu Hu, and Weisong Shi. Evaluation of JIAJIA software DSM system on high performance computer architectures. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, 1999.

[27] Ben Leslie et al. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[28] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[29] Ian Pratt et al. Xen 3.0 and the art of virtualization. *Proceedings of the Ottawa Linux Symposium*, 2:65–77, 2005.

[30] Manish Shah et al. UltraSPARC T2: A highly-threaded, power-efficient, SPARC SOC. In *Proceedings of the 2007 IEEE Asian Solid-State Circuit Conference*, 2007.

[31] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on OS Principles*, pages 211–223, 1989.

[32] Vincent W. Freeh, David K. Lowenthal, , and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 201–212, 1994.

[33] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, 1999.

[34] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessey. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Computer Architecture*, pages 15–26, 1990.

[35] Phillip Gibbons and Ephraim Korach. The complexity of sequential consistency. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 317–325, 1992.

[36] Robert Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, Cambridge, MA, 1972.

[37] Ganesh Gopalakrishnan, Dilip Khandekar, Ravi Kuramkote, and Ratan Nalumasu. Case studies in symbolic model checking. Technical Report UUCS-94-009, Dept of Computer Science, University of Utah, 1994.

[38] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on OS Principles*, pages 154–169, 1999.

[39] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 349–356, 1997.

[40] Zhiyi Huang, Wenguang Chen, Martin Purvis, and Weimin Zheng. Vodca: View-oriented, distributed, cluster-based approach to parallel computing. In *Proceedings of the 6th International Symposium on Cluster Computing and the Grid*, 2001.

[41] Zhiyi Huang, Chengzheng Sun, Stephen Cranefield, , and Martin Purvis. View-based consistency and its implementation. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, pages 74–81, 2001.

[42] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 39–50, 1993.

[43] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 277–287, 1996.

[44] Liviu Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, Dept of Computer Science, 1998.

[45] InfiniBand Trade Association. Infiniband technology overview. http://www.infinibandta.org/about/, accessed on 30th September 2008.

[46] Intel Corp. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, October 2002. http://www.intel.com/design/itanium2/documentation.htm.

[47] Intel Corp. *Itanium Architecture Software Developer's Manual*, January 2006. http://www.intel.com/design/itanium2/documentation.htm.

[48] Intel Corp. *Intel 64 and IA-32 Architecture Software Developer's Manual*, August 2007. http://www.intel.com/products/processor/manuals/index.htm.

[49] Kenji Kaneda. Virtual machine monitor for providing a single system image. URL http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/dvm/.

[50] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[51] Peter Keleher, Alan L. Cox, and Willie Zwanepoel. Lazy consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21. ACM/IEEE, 1992.

[52] R.E. Kessler and Miron Livny. An analysis of distributed shared memory algorithms. In *Proceedings of the 9th IEEE International Conference on Distributed Computing Systems*, pages 498–505, 1989.

[53] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 1–15, 2005.

[54] Maxim Krasnyansky and Maksim Yevmenkin. Universal TUN/TAP device driver. See Documentation/networking/tuntap.txt in a Linux source tree.

[55] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[56] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–1, 1979.

[57] Kevin Lawton. Running multiple operating systems concurrently on an IA32 PC using virtualization techniques, November 1999. docs/txt/paper-19991129a.txt in Plex86 source tree.

[58] Ben Leslie, Nicholas FitzRoy-Dale, and Gernot Heiser. Encapsulated user-level device drivers in the Mungi operating system. In *Proceedings of the Workshop on Object Systems and Software Architectures 2004*, Victor Harbor, South Australia, Australia, January 2004.

[59] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, April 2005.

[60] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report PA005520, National ICT Australia, October 2005.

[61] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7:321–59, 1989.

[62] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, 1997.

[63] Daniel J. Magenheimer and Thomas W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 73–82, 2004.

[64] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[65] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.

[66] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, October 1998.

[67] Cameron McNairy and Rohit Bhatia. Montecito: a dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[68] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Technical Conference*, pages 279–294, 1996.

[69] Message Passing Interface Forum. MPI: A message-passing interface standard, November 2003.

[70] Masaaki Mizuno, Michel Raynal, and James Z. Zhou. Sequential consistency in distributed systems: Theory and implementation. Technical Report RR-2347, INRIA, France, March 1995.

[71] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, and I.D. Scherson. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 277–286, 2004.

[72] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[73] Frank Mueller. On the design and implementation of DSM-Threads. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, 1997.

[74] Myricom, Inc. Myri-10G overview, December 2006. http://www.myri.com/Myri-10G/.

[75] MySQL AB. MySQL database software. http://www.mysql.com/.

[76] National ICT Australia. Kenge libraries. http://www.ertos.nicta.com.au/software/kenge/, accessed on 30th September 2008.

[77] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(8):52–60, August 1991.

[78] Martin Pool. distcc, a fast free distributed compiler. In *Proceedings of the 5th Linux.Conf.Au*, January 2004. http://distcc.samba.org/.

[79] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

[80] PostgreSQL Global Development Group. PostgreSQL database software. http://www.postgresql.org/.

[81] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in distributed shared memories. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 125–130, Dijon, France, 1996.

[82] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 340–347, 1984.

[83] Stephen Shankland. Virtualization start-ups hit reset button. CNET News.com, 2nd April 2006. http://www.news.com/Virtualization-start-ups-hit-reset-button/2100-7346_3-6056673.html, accessed on 30th September 2008.

[84] SPARC International Inc., Menlo Park, CA, USA. *The SPARC Architecture Manual, Version 9*, 2000. http://www.sparc.org/standards.html.

[85] Evan Speight and John K. Bennett. Brazos: A third generation DSM system. In *Proceedings of the USENIX Windows NT Workshop*, pages 95–106, 1997.

[86] Transaction Processing Council. TPC Benchmark C standard specification, June 2007. Edited by François Raab.

[87] Melinda Varian. VM and the VM community: Past, present, and future. SHARE 89 Sessions 9059–61, 1997. Available from http://www.princeton.edu/~melinda/.

[88] Alex Vasilevsky. Linux virtualization on Virtual Iron VFe. In *Proceedings of the Ottawa Linux Symposium*, July 2005.

[89] Bruce J. Walker. Open single system image (openSSI) Linux cluster project. http://www.openssi.org/ssi-intro.pdf, accessed on 30th September 2008.

[90] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[91] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[92] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software write detection for a distributed shared memory. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, 1994.