MagiXen: Combining Binary Translation and Virtualization

Matthew Chapman, Daniel J. Magenheimer, Parthasarathy Ranganathan
Enterprise Systems and Software Laboratory
HP Laboratories Palo Alto
HPL-2007-77
May 4, 2007*

virtualization, dynamic binary translation, hypervisor, Xen Itanium, virtual appliances, virtual machine monitors

Virtualization is emerging as an important technology in future systems, providing an extra layer of abstraction between the hardware and operating system. Previous work on virtualization has focused on the partitioning, isolation, and encapsulation features of virtual machines and their use for different applications, but mainly in the context of a specific processor architecture. In this paper, we argue for integrating an interface transformation layer to virtualization, specifically combining virtualization with a dynamic binary translator. This feature significantly increases the benefits from current applications of virtualization (e.g., for server consolidation and resource provisioning) while potentially enabling additional new uses of virtualization matched with emerging trends (e.g., virtual appliances and heterogeneous hardware). We have built MagiXen — pronounced “magician”— a prototype implementation of a Xen virtual machine monitor with integrated binary translation that can run IA-32 virtual machines on Itanium platforms. We present performance results for several typical benchmarks and discuss insights from our experiences with building the prototype.

* Internal Accession Date Only
Approved for External Publication
© Copyright 2007 Hewlett-Packard Development Company, L.P.

MagiXen: Combining Binary Translation and Virtualization

Matthew Chapman
matthewc@cse.unsw.edu.au

Daniel J. Magenheimer
Dan.Magenheimer@hp.com

Parthasarathy Ranganathan
Partha.Ranganathan@hp.com

University of New South Wales, Australia

HP Labs, Palo Alto, CA 94304

Abstract

Virtualization is emerging as an important technology in future systems, providing an extra layer of abstraction between the hardware and operating system. Previous work on virtualization has focused on the partitioning, isolation, and encapsulation features of virtual machines and their use for different applications, but mainly in the context of a specific processor architecture. In this paper, we argue for integrating an interface transformation layer to virtualization, specifically combining virtualization with a dynamic binary translator. This feature significantly increases the benefits from current applications of virtualization (e.g., for server consolidation and resource provisioning) while potentially enabling additional new uses of virtualization matched with emerging trends (e.g., virtual appliances and heterogeneous hardware). We have built MagiXen *— pronounced "magician" — a prototype implementation of a Xen virtual machine monitor with integrated binary translation that can run IA-32 virtual machines on Itanium platforms. We present performance results for several typical benchmarks and discuss insights from our experiences with building the prototype.*

1 Introduction

Virtual machine technology is rapidly emerging to become an integral component of future systems. A number of software-based virtualization solutions are commercially available (such as from VMware, Microsoft and XenSource), while Intel and AMD are introducing a series of platform changes to better support hardware-based virtualization. Mirroring this growth is a large and expanding body of academic, commercial, and open-source work in the area. Several recent studies [16, 15] report that 40–60% of enterprises already implement some form of server virtualization, and that within a few years, this fraction is likely to approach 100%.

Virtual machine technology provides three primary features. It allows multiple operating enviroments to reside on the same physical machine; it provides fault and security isolation which can be tied to guaranteed service levels; and it allows the entire state of the "machine" — memory, disk images, I/O state, etc. — to be captured, saved, and potentially reused. In turn, these three features of virtualization enable several interesting applications such as server consolidation, resource provisioning, software delivery, security, availability, debugging, simulation, etc. [30]

In this paper, we argue for adding one additional capability to virtualization, namely *dynamic transformation of the hardware interface*. Specifically, we propose combining dynamic binary translation of the instruction-set architecture with conventional virtualization functionality. This will significantly increase the benefits from current applications of virtualization while potentially enabling additional new uses of virtualization targeted at emerging trends in hardware and software.

For example, the traditional resource consolidation advantages of virtualization can be further improved by being able to consolidate enterprise deployments across different vendor ISAs — such as IA-32 (x86), Itanium, Power and SPARC — into one single server, but without the complexities of moving away from legacy environments. Similarly, the resource management benefits from VM migration can be enhanced through the increased diversity of heterogeneous server configurations to choose from.

Furthermore, the notion of a "virtual appliance" has been gathering momentum. A virtual appliance is an encapsulation of the application and operating system bits required for a workload, targeted at a particular virtual machine platform. Having cross-ISA translation at this level enables interesting new system architectures to be designed with a degree of freedom in the ISA that can be very valuable. It has also been suggested that virtual appliances can potentially lead to adoption of "custom operating systems" [28]. The consequent non-uniformity at the application binary interface (ABI) and the relative stability of the ISA interface motivate binary translation to be performed at this layer to continue to obtain all the benefits of binary translation.

From a hardware perspective, several academic studies have documented the benefits from asymmetric multiprocessing on a single core [19, 17, 5, 20]. Recent industry trends such as IBM's Cell processor and AMD's acquisition of ATI, and keynote talks from processor vendors [3, 26], suggest that this can be an interesting design point commercially. Having cross-ISA virtual machines can provide novel opportunities to leverage such asymmetry and design even more heterogeneity for improved performance. Finally, given the pervasiveness of

the IA-32 ISA, our approach will enable the IA-32 instruction set to transition from a hardware interface to essentially become an application that needs to be supported for future architectures; this can allow new server designs that go beyond the limited choices available from a small number of processor vendors.

There has been a lot of work on system virtual machines and binary translation individually, but we are not aware of any previous study considering the combination of both in the manner we suggest. Specifically, prior work on binary translation has predominantly focused at the application level (e.g., IA32-EL [7], FX!32 [11], QuickTransit [36]) or at the hardware level (Transmeta Crusoe processor [13], IBM AS/400). The closest related work we are aware of is the VirtualPC work [37] that uses binary translation to emulate a Windows system on a Mac platform, and the full-system mode of the popular Linux-based QEMU [10], but neither system is integrated with general virtualization software in the way we propose.

This paper discusses the implementation of virtual machines that augment traditional virtualization capabilities with integrated dynamic binary translation functionality. To understand the issues involved, we built MagiXen, an extension to the Xen virtual machine monitor that incorporates binary translation from the IA-32 instruction set to the Itanium instruction set. Given the large body of existing work on user-level binary translation, we focus primarily on system-level binary translation issues. Consequently, our approach leverages a commercially available binary translator, wraps this in a home-grown support layer for system ISA translation, and interfaces the combination with the open-source Xen (specifically Xen/ia64) virtual machine monitor.

We have built a prototype that currently runs certain IA-32 virtual machines on an Itanium machine. Key issues that we needed to address included interfacing to the Xen API, pagetable and segmentation handling, and Xen virtual I/O. Our performance results show that, for predominantly user mode benchmarks, the performance overhead of integrating binary translation at the virtual machine layer is similar to binary translation at the user level, while enabling all the benefits discussed earlier. For more OS-intensive benchmarks, our current implementation suffers from higher performance degradation. We evaluate the reasons for this higher overhead and discuss how these can be addressed. Overall, our experience with the prototype demonstrates that a cross-ISA virtual machine monitor with integrated binary translator can be a viable solution for the future.

The rest of the paper is organized as follows. Section 2 provides some background and further examines the motivation for a cross-ISA virtual machine. Sections 3 and 4 present our MagiXen prototype and our performance results. Section 5 discusses the insights from our prototyping effort and Section 6 summarizes some related work. Finally, Section 7 concludes the paper.

2 Background and Motivation

Although literature describing efforts in both virtualization and dynamic binary translation has been common in recent years, readers less familiar with recent advances will benefit from a review of some of the key concepts. In this section, we first review some general computer architecture terms fundamental to both fields, then review concepts related to virtualization, then dynamic binary translation. We then introduce our approach to combine the two and discuss the benefits from the combination.

2.1 Basic Concepts

An *instruction set architecture* (ISA) defines the boundary between machine hardware and software, and consists of a set of instructions and rules about how those instructions execute under various circumstances. On most machines, the hardware enforces some concept of *privilege* and software is executed in either *privileged mode* or *unprivileged (or user) mode*. Similarly, an ISA can be subdivided into two parts: the *system ISA* and the *user ISA*. In general, the system ISA consists of instructions that are privileged while the user ISA consists of instructions that are unprivileged, though there are some subtleties that we will expand on shortly. If software executing in user mode (an *application*) attempts to execute a privileged instruction, the hardware forbids the execution by invoking a *trap*, which delivers control to privileged code (generally an operating system). This allows an operating system to isolate applications from each other and to provide an illusion that each application owns all of the machine's resources (thus providing the foundation for *multi-programming*). Since an application may wish to control the machine in ways that are privileged, an operating system provides an abstract interface to privileged functionality through a set of *system calls*. Thus, an application may execute code either of the user ISA or system calls. The sum of these two is called an *ABI*.

2.2 Virtualization concepts.

Although the virtualization field admits to many different variations of implementation, from *container-based* virtual machines [31] to *hosted* virtual machines, we focus specifically on *hostless, system* virtual machines, in which a *virtual machine monitor (VMM)* enforces and abstracts the system ISA to support multiple operating systems — or *guest OSs* — on the same physical machine, each with their own set of applications. The VMM is also sometimes known as a *hypervisor*.

On some architectures, the system ISA and user ISA are clearly disjoint. On others, however, there is a subtle overlap. Specifically, there are privileged instruc-

tions that have one result when executed in privileged mode and a different — non-trapping — result when executed in unprivileged mode. Such instructions are called *privilege-sensitive* instructions, or *problem* instructions as opposed to all other instructions which are called *safe* instructions. If an ISA has no privilege-sensitive instructions and also conforms to certain other constraints [24], it is referred to as *classically virtualizable*. Notably, the Intel IA-32 ISA has privilege-sensitive instructions and thus is not classically virtualizable [27].

Whether or not an ISA is virtualizable is important because in order to enforce isolation, the VMM must be the most privileged software and so a guest operating system runs *deprivileged*. Just as an operating system would trap any attempt by an application to execute a privileged instruction, the VMM traps every privileged instruction executed by a guest OS. This *trap-and-emulate* technique is widely described in the literature and has proven effective at guaranteeing guest isolation but, depending on the workload, may have significant performance impact. Worse, trap-and-emulate fails if the ISA contains privilege-sensitive instructions; since these instructions do not trap, the VMM cannot enforce the different semantics intended when executed by a deprivileged OS.

As the Intel IA-32 architecture has become increasingly ubiquitous, various researchers have explored how to overcome the hazards resulting from the fact that IA-32 is not classically virtualizable, while balancing the tradeoffs of faithfully emulating the IA-32 ISA with the performance loss of trap-and-emulate. VMware [33] provides virtualization of unmodified guests using a technique referred to as *binary rewriting*, which examines each instruction prior to execution and translates problem instructions into safe instructions, with some cost in runtime performance. Paravirtualization is a technique first introduced by Denali [42] and popularized by Xen [8], whereby the guest OS is modified to cooperate with the VMM; problem instructions are manually converted to safe instructions in the guest OS source code and *hypercalls* (hypervisor calls) are added to communicate efficiently with the VMM. In essence, the strict definition of the ISA is blurred in order to obtain performance. Although the impact of paravirtualization on the guest OS can be minimized [21], use of the approach is largely limited to OS's for which source code is available. More recently, Intel has added a set of architectural extensions called Intel Virtualization Technology (VT) [38] to the IA-32 ISA which allow classical virtualization to be supported; AMD has an analogous scheme called AMD Virtualization (more commonly known by the codename Pacifica). While one might assume that the VT hardware solution has eliminated the need for software solutions, studies [1] have shown that the increased dependency of VT on trap-and-emulate results in a significant relative performance loss for many workloads.

Project	Source ISA	Target ISA	OS
HP3000OCT	HP3000	PA-RISC	MPE
FX!32	IA-32	Alpha	WinNT
Aries	PA-RISC	Itanium	HP-UX
IA-32 EL	IA-32	Itanium	Win or Linux

Figure 1: Some commercial application-level dynamic binary translators

Regardless of the technique used, modern virtualization has enabled a number of exciting new capabilities. An entire guest OS and all the applications running on it can be stopped, saved, and then restored and restarted on a different physical machine [29]. This migration may even be conducted live [12, 39], such that the apparent time the guest is stopped is so short that it is essentially zero to an outside observer. Live migration has sparked research toward optimizing data center resource utilization, while the ability to encapsulate an entire operating environment into a distributable file has enabled a nascent market for virtual appliances [40]. This in turn has engendered speculation that the role of the future general purpose operating system will be greatly diminished [28] or perhaps even eliminated [9]. We will expand on these benefits shortly.

2.3 Dynamic binary translation concepts

Dynamic binary translation (DBT), as its name implies, is runtime compilation of a binary image from one architecture so that the resultant code will run on another architecture. The translation occurs between a *source* environment and a *target* environment. The exact characteristics of both the source and target environments yields a variety of types of translators, which we will briefly taxonomize with examples.

In recent decades, many new machine architectures have been introduced, and commercial success of a new machine is often dependent on the number and breadth of applications the new machine is capable of running. Consequently, hardware vendors have invested a great deal of engineering and money in tools that assist and encourage software vendors to port their applications to new architectures, and even more in ensuring that the porting process is easy and that the applications have both high fidelity and high performance. As a result, commercial *application-level binary translators* are the most prevalent and the most commercially-stable form of DBT. A few commonly cited in the literature are shown in Figure 1, along with the source and target ISAs and OS.

Recall that an ABI consists of two interfaces, a user ISA and a set of system calls associated with a particular OS. All of the examples in Figure 1 translate from one ISA to another but retain the same OS. We call

these *cross-ISA-same-OS* translators. QuickTransit [36], a product of Transitive Corp, supports multiple source and target ISAs but also allows the source and target OS to differ; we call this a *cross-ISA-cross-OS* translator. We explicitly reemphasize that, although we use the term ISA in both of these DBT categories, it is specifically the user ISA that is being translated; none of these examples translate the system ISA.

We note in passing that there are also *same-ISA-cross-OS* translators, such as Wine [35] which allows many unmodified Windows programs to run on several IA-32 Unix flavors, Project Janus [34] which provides an environment for running Linux applications on IA-32 Solaris, and the not-subtly-named Linux Runtime Environment for HP-UX [18] available on HP's Itanium machines. And to fill out the matrix there are even *same-ISA-same-OS* translators, such as Dynamo [6], in which the translation is solely focused on improving performance.

However, the DBT category most relevant to our discussion are those that translate between full ISAs, not just the user ISA but also the system ISA. These DBTs are generally disguised as part of the hardware to create the illusion that a machine implementing one ISA actually supports a completely different ISA. This hardware generally implements a fundamentally different architecture intended to provide some market differentiation, such as dramatically greater performance or reduced power consumption. The most notable commercial example is Transmeta's code-morphing software (CMS) [13] which combines an interpreter, optimizing DBT, and runtime system, to allow a co-designed VLIW processor with little resemblance to IA-32 to nevertheless boot and run unchanged IA-32 operating systems and applications.

Translating efficiently and reliably between ISAs presents some significant challenges and engineering tradeoffs, many of which are enumerated in detail in [4]. We have encountered a number of these challenges firsthand and will explore some of them in later sections.

2.4 Proposed Approach

While virtualization research is experiencing rapid growth, DBT research is presently in decline. This is in part due to the rise in popularity of the IA-32 architecture and the resultant reduction in ISA diversity in the computer marketplace. New machine architectures have been relegated to smaller but still significant computing niches, such as high-performance and fault-tolerant machines at one extreme, and game consoles and smartphones at the other. Some believe that the IA-32 ISA will continue to dominate and, indeed, eventually become the only architecture of interest; others believe that monopolies historically tend to unleash countervailing innovations. Regardless of one's crystal ball, for the near future it is certain that IA-32 will be a dominant force and existing and emerging machines of other architectures would do well to harness this force. Consequently, we resonate with this quote from the work of the DynamoRIO team [41]: *In the future, legacy ISAs such as x86 will no longer simply be an ISA, but rather x86 and the accompanying ecosystem will become an application that all future architectures will have to execute effectively.*

To achieve just that, we propose to combine the inherent flexibility and emerging applications of a hostless system VMM with the transformational capability of a cross-full-ISA DBT. Our approach provides several benefits which are discussed below.

More potential for resource consolidation. One of the key customer usage models for virtualization today is consolidation, where the workload of several smaller servers is combined into one larger server. Currently, the workloads to be consolidated are platform-specific which constrains the choices for purchasing the new server. For example, if the workload contains applications that only run on Windows Server on an IA-32 platform, the consolidation server must be capable of running IA-32 Windows applications. However, the selection of "large servers" on the market consists largely of systems built on other ISAs such as Itanium, Power, and SPARC. Though Windows Server supports Itanium, many IA-32 Windows applications have never been ported to Itanium. Our approach allows the workload of both IA-32 servers and non-IA-32 servers to be consolidated onto more high-end non-IA-32 servers.

Resource migration. Another key value of virtualization is in disconnecting the virtual platform from the physical platform. A workload can be transparently and dynamically moved from one platform to another, even without reboot or noticeable downtime. This live migration provides great flexibility: workloads can be moved to maximize resource utilization, allow for maintenance, reduce power and cooling costs. But one of the not-widely-advertised limitations of migration is that the source and target processors must be identical, or nearly so. Certainly the ISA must be the same, but even to the extent that no currently available virtualization software allows for Pentium 4 applications to run on a Pentium 3. Our approach allows resource pools with completely different ISAs to be combined into the same pool. Our technique could also be used to combine non-identical similar-ISA pools (such as the example of P4 vs P3). Even more aggressively, having a cross-ISA virtual machine migration provides a level of ISA agnosticity that can enable more power-efficient or lower cost general-purpose processors from different markets (e.g., SiByte, PASemi) to support a broader set of applications.

Emerging virtual appliances and ISA agnosticity. An interesting and rapidly growing niche of the virtualization market is the availability of "virtual appliances",

packages of bits combining an OS and an application that are distributed to run on top of a particular virtual machine monitor product. This greatly reduces installation, hardware, and testing costs for sampling new software. Some believe that its advantages are sufficient that it could replace current software distribution models. However, creating virtual appliances involves some of the same vendor costs as traditional software delivery, and thus virtual appliances are likely to be first (or only) available on the dominant ISA and virtualization platforms, e.g. VMware on IA-32. In such cases, our approach could allow commercially available IA-32 virtual appliances to run on other platforms.

Custom operating systems. Some believe that the advent of virtual appliances will hasten the end of the reign of the classic OS at the center of the software universe [28]. For example, most applications run on Windows because Windows is nearly ubuquitous; nearly all application vendors' customers are already running on Windows, so the barrier for deployment of new applications on Windows is small, which in turn makes Windows even more ubiquitous. As a result of this virtuous (for Microsoft) cycle, the majority of hardware and software is designed for Windows. However,virtualization allows multiple software stacks to coexist, so application vendors are free to build a stack using different criteria. For example, Oracle could deliver a Linux-based database environment that could run on a server that is simultaneously running applications in a Windows environment. Indeed, some application vendors are considering elimination of the OS entirely [9]. Should this trend toward OS-agnosticity gather momentum, traditional application-level binary translation software, which is very dependent on translating between the ABIs of a rapidly diversifying set of OS's, could be at a severe disadvantage. Our approach is much more conducive to this trend as our nexus of translation is at the ISA which, due to the huge costs of changing hardware, is much more stable.

Heterogeneous hardware. Looking even further, another trend has been the support for heterogeneous or asymmetric cores in future processors. For example, AMD's Torrenza [2] initiative allows for heterogeneity at the socket level that can enable, say, a graphics processor to co-exist with a general-purpose processor. Other studies have also discussed heterogeneity at the core level [19, 17, 5, 20]. In these environments, having a binary translation layer combined with the virtual machine layer can enable future virtual appliances to be seamlessly migrated between configurations supporting different heterogeneous accelerator cores.

In short, we believe that the combination of virtualization with binary translation provides the advantages of both. However, each technology presents significant challenges too and these challenges could diminish the benefits. To test this, we implemented a prototype, which we now introduce. We call our prototype MagiXen (pronounced "magician").

3 MagiXen Prototype Design

In this section, we discuss the design of MagiXen, our prototype virtual machine monitor with integrated dynamic binary translation. MagiXen augments an application-level binary translator with a system ISA translation support wrapper, and then mates it with a system virtual machine monitor. More simply, MagiXen allows virtual machines of one architecture to run on another architecture, and more specifically allows IA-32 paravirtualized virtual machines to run alongside Itanium virtual machines on an Itanium VMM.

3.1 MagiXen implementation overview

As described earlier, there has been a large body of prior work on user-level binary translation. Thus, for our prototype, we decided to leverage an existing binary translator: IA-32 EL (Execution Layer) from Intel [7]. We also considered QEMU [10], but the performance of QEMU on translated user code does not compare favorably with commercial translators like IA-32 EL. The latter is very well optimised for the Itanium target platform, and uses a two-stage translation process: a fast template-based translation stage with some instrumentation, and a re-optimisation stage using the results of the instrumentation, for frequently used code. Further, IA32-EL is commercially available for several operating systems, has proven fidelity and is supported by Intel. In contrast, QEMU translation is purely template-based, and the Itanium support is largely orphaned and unstable. Since we consider both stability and performance to be important parts of validating the approach, we decided to use IA-32 EL. The main disadvantage of using IA-32 EL is that it is not open source; the source is closely guarded by Intel. It had to be regarded as a black box; the interfaces to it were gleaned from the shim code shipping with Linux distributions.

Figure 2 shows the high-level organization of the prototype. We architect our MagiXen prototype using the Itanium port of the open-source Xen hypervisor (Xen/ia64) [25] as a foundation, and IA-32 EL for application-level ISA translation, but developed our own home-grown transformation layer to handle system-level instructions and emulate Xen APIs.

The core binary translation functions are provided by the Intel IA-32 EL component, which can be considered a software implementation of the user IA-32 ISA. MagiXen acts like a virtual machine monitor, providing an emulated IA-32 Xen environment for the guest kernel, in terms of the Itanium APIs provided by Xen/ia64.

For the initial prototype, we chose to require a guest

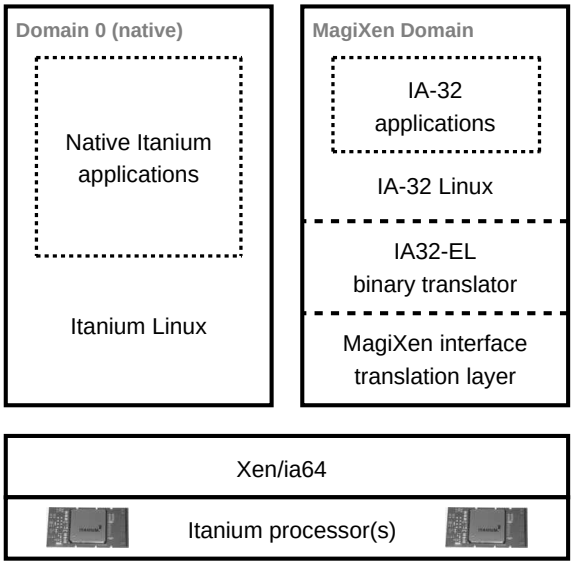

Figure 2: MagiXen architecture

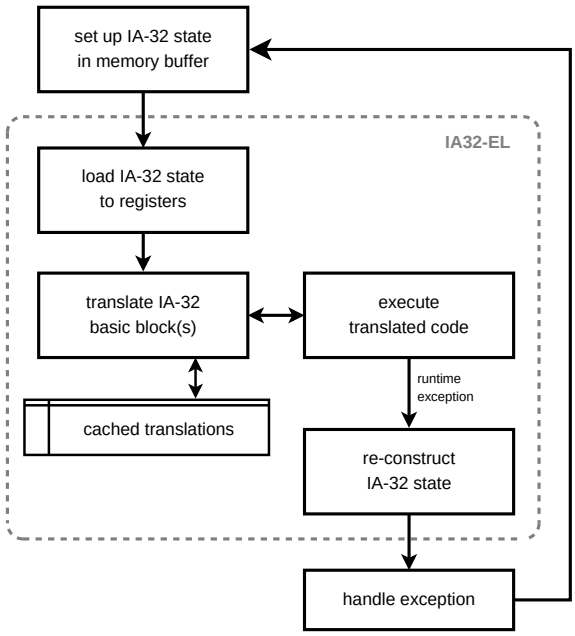

Figure 3: IA-32 EL execution flow

kernel that has been paravirtualized for Xen, rather than an unmodified stock kernel compiled for a hardware platform, for a number of reasons.

Firstly, we consider the most common use case to be migrating a complete virtual machine image from an IA-32 Xen hypervisor to an Itanium Xen hypervisor; whether done manually or through automatic load balancing. Migrating an image from real IA-32 hardware to Xen is feasible, but is a much more difficult operation. For example, the same devices may not be available.

A kernel built for Xen also has a simpler boot sequence and virtualized drivers. If simulating a real hardware platform, one would need to create dummy firmware tables and emulate real devices, adding significant complexity.

Finally, as previously noted, the IA-32 architecture is not classically virtualizable. In particular, there are a number of privilege-sensitive instructions which do not generate traps but need to be intercepted by a virtual machine monitor. In theory this should not be a problem for dynamic translation; however currently, like real IA-32 hardware, IA-32 EL does not produce traps for these sensitive instructions. This behaviour is difficult to change since the IA-32 EL source is not publicly available. When using a paravirtualized kernel built for Xen, any privilege-sensitive instructions have already been replaced, which circumvents the problem.

While this decision limits the ability for the MagiXen prototype to support commercially available virtual appliances, future versions of popular operating systems are expected to support transparent paravirtualization and thus run both natively and as a Xen guest. Virtual appliances based on such dual-capable operating systems could be supported in MagiXen.

3.2 Startup

In keeping with the desire to minimize Xen modifications, MagiXen is packaged into a loader binary which uses the Linux boot protocol, so that it can be started in the same way as a Linux domain (a *domain* is the Xen terminology for an individual virtual machine). The loader binary is specified instead of an Itanium Linux kernel, and the IA-32 guest kernel is specified in place of an initial RAM disk.

The loader binary contains the MagiXen runtime component as well as the Intel IA-32 EL component. The loader relocates the components to their runtime locations, establishes virtual mappings, switches to virtual mode, and starts MagiXen. The loader memory is then no longer needed and can be reclaimed.

MagiXen is responsible for installing an *interruption vector table*, a table of entrypoints for handling Itanium processor exceptions generated by the domain. It also sets up an initial IA-32 environment, including register state, initial pagetables, and a memory structure containing Xen-specific startup information for the guest operating system.

MagiXen then passes control to IA-32 EL's fetch-and-execute loop, as shown in Figure 3. Control is returned to MagiXen only when one of three things happens:

- IA-32 EL encounters an exception condition such as a privileged instruction or hypercall; IA-32 state is saved and a MagiXen callback function is invoked.
- Execution causes a hardware exception such as a

page fault, which is reflected by Xen to MagiXen. Typically execution resumes at the interrupted point after handling. However, if an exception needs to be delivered to the guest OS, MagiXen calls back to IA-32 EL to re-construct IA-32 state. It then modifies the state to simulate exception delivery, and restarts the fetch-and-execute loop.

- Execution is interrupted by an asynchronous notification destined for the guest operating system, such as a device interrupt. MagiXen checks if the notification is safe to deliver immediately. If so, it proceeds as for an exception; otherwise, it requests IA-32 EL to halt execution as soon as possible, and resumes execution of IA32-EL, which invokes a MagiXen callback at the next safe point.

3.3 Emulating the Xen API

In accordance with the Xen API, the guest kernel performs Xen hypercalls by branching to offsets within a special page. That page is filled out by the hypervisor, or MagiXen in this case, at startup.

In general, hypercalls need to exit IA-32 execution and call back into MagiXen for handling. Any IA-32 instruction that exits execution should be equally efficient; we use a software interrupt instruction, just like real Xen. MagiXen then simply inspects IA-32 register state and dispatches to the appropriate emulation function.

There are around 20 different hypercalls used by an IA-32 guest kernel, and many of these have more than one sub-operation. Even so, implementing the emulation for these was not especially difficult. Some of these hypercalls are emulated by calling the corresponding Itanium APIs, while others act purely on data maintained within MagiXen.

MagiXen also emulates a number of privileged instructions which are still present in a kernel compiled for Xen. These include `iret` (return from interrupt), I/O port accesses, and some control register accesses. This is implemented in a similar way to any other virtual machine monitor: by inspecting the instruction at the IA-32 instruction pointer, and updating IA-32 register state appropriately.

3.4 Segmentation

For performance reasons, IA-32 EL assumes a simple segmentation model, with the main code, data and stack segment selectors (`cs`/`ds`/`ss`) accessing a flat 32-bit linear address space starting at address 0. In other words, segmentation is ignored in the common case, and IA-32 references to a given address are translated into Itanium references to the same address. Otherwise, every memory reference would need to be instrumented with code to add the base address of the segment and to check limits and permissions.

Fortunately, this is also sufficient for kernel code; most modern operating systems including Linux use such a flat memory model, and only ever use a base address of 0 in these segment descriptors. The lack of limit checking is not without side-effects, however, since Linux relies on these limit checks to protect the kernel from user code. Instead we must use paging mechanisms to provide this protection.

The other segment selectors (`es`/`fs`/`gs`) are often used for special purposes; in particular `gs` is typically used on Linux for thread local storage. IA-32 EL implements these more fully, with a configurable base for each and a callback to MagiXen when the selectors are modified, thus MagiXen can correctly emulate such accesses.

3.5 Paging

Since the IA-32 linear address space must start at address 0, the bottom region of the Itanium virtual address space is set aside for IA-32 virtual mappings. Within this region, 4KB mappings are established, corresponding directly to the mappings that would be present in the TLB of an IA-32 processor. (In order to allow 4KB mappings, Itanium Xen must be configured for 4KB page size internally.)

On an IA-32 system, when a TLB miss is encountered, the IA-32 processor directly accesses the operating system pagetable, allowing the miss to be resolved in around 50 cycles. This is even true when running on Xen; one of Xen's novel features is that it sets up the hardware to access the domain's pagetable directly (and monitors pagetable writes to prevent subversion) [8].

On an Itanium system with MagiXen, however, the Itanium processor is not capable of interpreting the domain's IA-32 pagetable. Instead, a fault is delivered to Xen, which is reflected to MagiXen, which accesses the IA-32 pagetable and provides the required translation. This is significantly more expensive, to the tune of around 1900 cycles. In order to improve this situation, we implemented an IA-32 pagetable walker within Xen, so that most faults can be handled within the Xen hypervisor without needing to be reflected to MagiXen. However, the overhead of TLB misses is still not negligible — around 900 cycles.

This is compounded by the fact the IA-32 architecture necessitates a TLB flush for every context switch (whenever the pagetable base register is changed). It is not possible to avoid this flush since existing operating systems depend on this behaviour. After such a flush, there is a flurry of TLB misses as the required translations are established; thus, on a system where TLB misses are expensive, the indirect cost of a context switch is also high.

Even worse, flushing translations also necessitates flushing the corresponding pre-translated code. If this is not done, the old code on the page will still be looked up and executed by the execution engine, even if the page is no longer mapped. Thus, after a full TLB flush, IA-

32 EL starts with a cold translation cache, and must re-translate the IA-32 code.

Ideally, the old translated basic blocks could be cached, and later restored if the same TLB entry is re-established. Unfortunately such a scheme is difficult to implement in the present system because IA-32 EL is closed source.

However, since MagiXen knows the location of the guest kernel, and that code can be assumed not to change, a simple optimisation is to avoid flushing kernel code on a context switch. Additionally, we can disable self-modifying-code detection for the kernel area. This results in a modest performance improvement.

3.6 Devices

The Xen hypervisor does not contain drivers. Instead, device drivers are contained within certain privileged driver domains, such as Domain 0 (the first domain started when booting Xen). These domains then service I/O requests on behalf of unprivileged domains, using inter-domain communication abstractions provided by Xen.

Most architecture-independent Xen abstractions, particularly those used for inter-domain communication, are mapped by MagiXen directly onto those provided by the host hypervisor. This allows stub drivers in the IA-32 guest operating system to communicate with real hardware drivers running in a driver domain, via the normal Xen mechanisms, with minimal intervention from MagiXen.

For example, event channels are used for asynchronous notifications between domains in Xen. MagiXen propagates event channel requests to the host hypervisor and delivers events from the hypervisor to the guest operating system. One example is the timer interrupt, which is delivered as an event like any other.

Another Xen abstraction is the grant table, used to share pages between domains. The granting domain puts an entry into its grant table specifying the physical frame number and destination domain; the receiving domain asks Xen to map the page by supplying the source domain and grant table index. Thus, normally there are no hypercalls necessary on the granting side.

Originally, we had placed MagiXen at the bottom of the domain's memory, and offset the guest's physical memory by a fixed amount. However, this meant that the guest's physical frame numbers did not correspond to real physical frame numbers, and thus hooks needed to be added to the guest kernel to translate entries and place them into the Itanium grant table.

To avoid this, we re-arranged the address space such that the guest's physical memory is at the bottom of the domain's physical memory, and MagiXen at the top, as per Figure 4. This allows the Itanium grant table to be exported directly to the guest kernel. The downside is

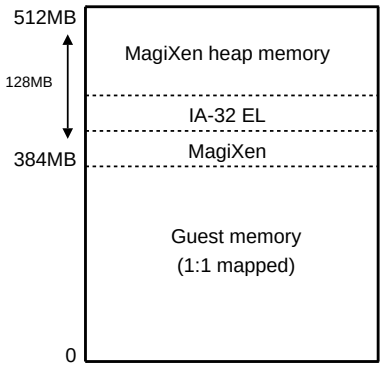

Figure 4: Memory layout of a 512MB domain

that the guest OS could specify MagiXen addresses in grant table entries, and thus MagiXen itself is not completely protected from the guest OS kernel. However, we consider this to be only a academic issue, since the guest OS would normally be privileged inside a domain in any case, and inter-domain protection is enforced by Xen.

An implication of sharing memory directly between IA-32 and Itanium domains is that the data structures used in that memory must be compatible. Endianness is not a problem for MagiXen, since both of the architectures are little-endian[1]. However, size and alignment needs to be considered carefully. We only found one problem in the Xen drivers, where a misaligned 64-bit value is aligned to a 64-bit boundary on Itanium and only a 32-bit boundary on IA-32. This can be addressed by making sure that the 64-bit value is always aligned. (A similar problem would arise if attempting to run an IA-32 guest on AMD64.)

Another issue that should be considered is that sharing pages with other domains could bypass self-modifying-code detection; if the guest kernel was to map a currently-mapped instruction page to a remote domain for DMA, the changed code might not become immediately visible to the instruction stream. However, this scenario is unlikely to happen in current operating systems.

3.7 Disabling the FPU

The IA-32 architecture provides a mechanism to temporarily disable the floating point unit, resulting in an exception when the FPU is accessed. Typically this is used to detect when a process needs to use the FPU, so that floating point state can be saved and restored only when necessary.

However, as far as we can tell IA-32 EL does not currently provide any such mechanism to disable IA-32 floating point instructions or access to IA-32 floating

[1]Technically, Itanium is bi-endian, but Xen/ia64 and Linux use little-endian mode.

point registers. Disabling Itanium floating point registers does not have the intended effect, since IA-32 EL also uses the Itanium floating point registers internally and for integer multiplication/division.

As a workaround, we signal this exception whenever returning to userspace with the floating point unit disabled. The net effect is that every process appears to use the FPU, whether or not it actually does. This guarantees correctness at the expense of performance, but the performance impact is barely noticable in our benchmarks.

4 Preliminary experiments

In this section, we present our performance results for two widely used benchmarks: AIM9 and SPEC CINT2000. Together these measure application-level computational performance as well as the overheads of various operating system mechanisms. In order to aid in understanding the overheads involved, we also microbenchmark some primitive operations such as system calls.

It is difficult to choose a good reference point for benchmarking a dynamic translation system. The most obvious option might be to compare against an IA-32 system; however, it is not clear what IA-32 system to choose to provide a fair comparison. Therefore, generally the most sensible comparison is with native code executing on the same system. We have normalised the performance numbers to the native benchmark performance on Itanium, in a native Xen domain. (Admittedly this does depend on the quality of the native compiler, but one could consider the compiler to be part of the platform, since most users are interested in the performance resulting from a combination of compiler and hardware.)

Our test system is an HP rx2600 Itanium server running a recent version of the Xen/ia64 hypervisor (specifically changeset 12895 of the 3.0.4 series, plus a number of minor fixes which we hope to submit to the Xen maintainers). The test system has two physical processors but it has been configured to use only one, to improve reproducibility of results. The processor is a 1500Mhz Itanium 2 "Madison" part, with 6MB of L3 cache. A single privileged domain (Domain 0) runs continuously to provide drivers for the real hardware. There are two unprivileged domains configured — one for MagiXen, and one for a second Itanium domain, each assigned 512MB of memory — but these are started as needed, and never run at the same time during benchmarking.

With the exception of the microbenchmarks, for which we used the processor cycle counter, the benchmarks all use the `gettimeofday` system call to measure wall-clock time. We verified that this system call provides good correspondence with wall-clock time both within the native domain and on MagiXen. SPECint automatically runs each test three times and uses the median; we have followed the same methodology for the AIM9 results.

	Xen/ia64	MagiXen
Null hypercall	430 cycles	800 cycles
Null system call	600 cycles	5500 cycles
Handle TLB miss	45 cycles (h/w)	900*/1900 cycles
Handle timer tick	6700 cycles	1000000+ cycles

* with modified Xen

Figure 5: Approximate costs for common operations

4.1 Microbenchmarks

Times for some basic operations are presented in Figure 5.We used `getppid` to represent a "null" system call — one that does little work — and `xen_version` to represent a "null" hypercall.

Generally we used a simple microbenchmark methodology: placing the code to be measured inside a loop that is executed a large number of times, subtracting the overhead of the loop by itself, and dividing by the number of iterations. We also varied the loop count to verify that the rate of increase produces a similar number. (The Itanium processor uses in-order execution, with stalls occuring when the results of long latency operations are consumed prematurely, so this methodology should be reasonably accurate providing that the results of any operations are consumed within the loop. This is not necessarily true on a real IA-32 processor, where loop iterations can interact.)

There is one IA-32 EL specific issue that we had to contend with. If a basic block is executed more than 4000 times (the *heating threshold*), the hot code optimizer is invoked to re-optimize the code. This optimization is very expensive (at least a million cycles in our experience) and naturally changes the timing of the code, disturbing any methodology that relies on each iteration having a fixed cost. Thus we were very careful not to run any particular instruction sequence too many times, using multiple loops where necessary.

For measuring timer overhead, we used a different strategy. We wrote a small program which polls the cycle counter in a loop, and notes any discontinuities more than 1000 cycles in a memory buffer. In this way, we obtained a trace of the occasions when application execution is interrupted. Most commonly this is due to the local domain's timer tick, although Domain 0's timer tick is also visible in the data, as well as some negligible noise due to other Domain 0 driver processing (such as incoming network frames). However the total overhead due to Domain 0 is not more than 0.2%.

A surprising result is that timer-related processing takes over a million cycles per tick (with large variance in this number); this time is primarily spent executing in translated kernel code, and some of it can be attributed to speculation-related TLB misses (see Section 5). While we have limited visibility into the inner

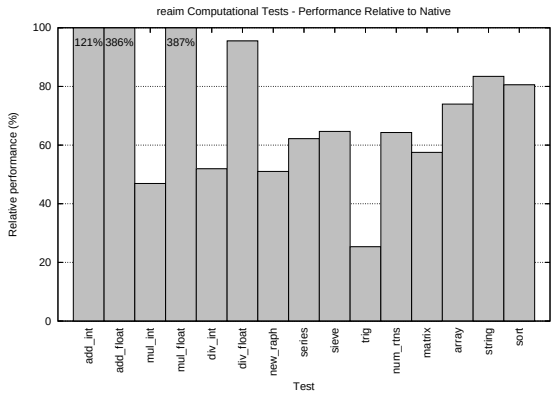

Figure 6: Results of `reaim` computational tests

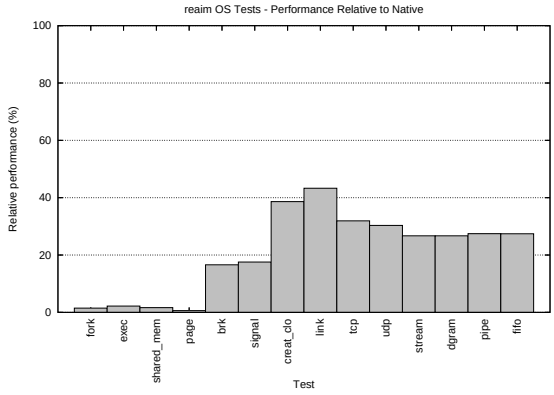

Figure 7: Results of `reaim` operating system tests

workings of IA-32 EL, we expect that the code in the timer interrupt would rapidly pass the heating threshold and be re-optimized. It is possible, then, that the optimizer uses speculation somewhat too aggressively for this code. Since the guest kernel is using a 100Hz timer, the effect is that as much as 7–8% of the available CPU time is spent executing the timer interrupt. In comparison, on the native domain, the local timer interrupt accounts for less than 0.1% of available time. This is clearly an area that should be addressed in future work.

4.2 AIM9

AIM9 is the short name for the AIM Independent Resource Benchmark Suite IX, originally from AIM Technology. The goal of AIM9 suite is to separately microbenchmark different aspects of a UNIX computer system, independently measuring processing rates for a large number of workloads including numerical operations, C library calls and UNIX system calls.

We use a modern re-implementation of AIM9 named `reaim` [23]. `reaim` is also capable of simulating the AIM Multiuser Benchmark, AIM7, however here we used the AIM9 mode.

The reaim workloads can roughly be divided into two categories: those which extensively use operating system services, such as I/O workloads, and those that do not, such as numerical workloads.

Figure 6 shows results for computational tests in the suite. These include simple microbenchmarks which add, multiply and divide numbers, as well as small practical codes such as finding roots of an equation (Newton-Raphson method), finding prime numbers (sieve), matrix multiplication, array sorting and string manipulation.

These numerical applications perform well, generally between 50-80% of native performance, which is a tribute to the IA-32 EL optimizer. In fact, some of the trivial test loops, particularly adding and multiplying floating point numbers, perform better on IA-32 EL than the native code compiled with gcc, thanks to the very good IA-32 EL optimiser. The one disappointing result is for the `trig` test, which benchmarks trigonometric functions in the system math library. The native math library contains handwritten (and very well tuned) assembly code; thus it is no great surprise that the translated code cannot match its performance.

Figure 7 shows results for OS-oriented workloads. Some of the tests are dominated by the cost of invoking various kernel services via system calls. These include `creat_close` and `link`, which perform file operations, `brk`, which uses the system call of the same name to resize its data segment, as well as `tcp`, `udp`, `stream`, `dgram`, `pipe` and `fifo`, which test various mechanisms for local inter-process communication (but are implemented within the one process). `signal`, which sends UNIX signals to itself, can also be considered to be in this category. The performance of such benchmarks is around 20–40call overhead is significantly higher in MagiXen, so this is not unreasonable.

The remaining four benchmarks, `exec`, `fork`, `shared_mem` and `page`, perform very poorly. The primary reason that the operations they perform — tearing down mappings, and in some cases context switching — cause TLB flushes in every iteration of the test loop. For the reasons mentioned previously, TLB flushes have a large indirect cost on MagiXen because of the large cost of re-establishing the mappings. In fact, `page` not only causes TLB flushes, but then deliberately causes extra page faults, pushing MagiXen's worst behaviour to its limits (the performance of `page` is around 0.6%, the others three around 2%). Fortunately, such extreme test loops are unlikely to occur in real applications, but they do highlight the limitations and are useful in understanding the behaviour of other benchmarks. We address possible improvements in Section 5.

4.3 SPEC CINT2000

SPEC CINT2000 [32] is a collection of applications representative of a compute-intensive workload with pri-

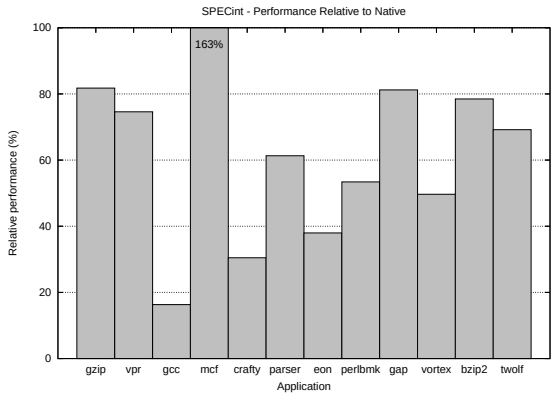

Figure 8: Results of SPEC CINT2000 benchmarks

marily integer computation (there is a CFP2000 suite for floating point applications). The CINT2000 applications vary in memory usage, but generally spend little time performing I/O compared to computation time.

Figure 8 shows the relative performance of the IA-32 CINT2000 applications on MagiXen, as compared to their native Itanium counterparts. The shape of the graph is essentially the same as that presented in the IA-32 EL paper [7]. This is to be expected, since these are computational benchmarks which depend on the performance of the application-level binary translator. The raw results vary slightly, some higher and some lower, however this can be partly attributed to the fact that we used a different compiler (gcc 3.3 rather than the proprietary Intel compiler). The high overhead of timer handling is also likely to degrade performance slightly.

The `mcf` benchmark is unique in that it actually performs better than the native version; the reason is that it uses word-sized data and thus the 32-bit version has a significantly smaller memory footprint (100MB vs 190MB) [32]. `gcc`, on the other hand, performs poorly. It is very memory intensive — even more so than `mcf` — and also maps and umaps memory in various stages of the run, causing TLB flushes and a large number of page faults.

With the exception of these two benchmarks, the performance is in the range of 30–80% of native, with a harmonic mean of 59% (compared to 61% for the IA-32 EL on Linux results — and yet MagiXen is virtualizing the whole operating system, not just a single application).

5 Discussion

Performance is very good for most of the numerical benchmarks, validating IA-32 EL, which has clearly been optimised for numerical workloads such as SPEC CINT2000. Memory and system call intensive workloads perform worse, although still within reasonable expectations. Workloads involving multiple processes and context switching perform poorly.

Profiling using Xenoprof [22] shows that, despite the optimisations described in the design section, a large portion of the time is still being spent servicing TLB misses.

Some of these TLB misses are to invalid addresses; the result of widespread use of control speculation within IA32-EL. However, making hardware defer TLB misses as late as possible only produces a small improvement on some benchmarks and a small degradation on others; mostly it just transfers some of the overhead to speculation recovery code. It is possible that some of the uses of speculation in IA32-EL are overly frivolous; effective use of control speculation requires that the speculative memory reference is valid in the vast majority of cases. Also, presumably IA-32 EL is optimised for Linux where TLB misses are less expensive than on Xen.

The I/O and system call intensive workloads suffer because all system calls and hypercalls require temporarily exiting the fetch-and-execute environment, and this is heavyweight in IA-32 EL, of the order of 800 cycles. This is because the full IA-32 state is saved and restored. While this provides a nice robust interface, it sacrifices performance when callbacks to MagiXen are quite frequent. It would be preferable if such callbacks could be made with the majority of IA-32 state still in registers, and the remainder of the state only saved on request. For example, in most cases there is no need to save and restore floating point state.

A large amount of overhead can also be accounted to context switches, and the fact that all translations are lost on every context switch, in terms of both TLB translations and also the pre-translated Itanium code. Implementing an IA-32 pagetable walker within the Xen hypervisor does improve the situation somewhat, since it reduces the cost of TLB misses when the translation is in the IA-32 pagetable, which is usually the case for the translations that need to be re-installed after a context switch. However, the cost is still a lot greater than a TLB miss on an IA-32 processor, and this does not address the indirect cost of having to re-translate the IA-32 code.

If one was allowed to modify the guest kernel for a slightly different paravirtualised architecture, a good optimisation for a system like MagiXen would be to introduce the concept of multiple contexts, as supported by most MMU architectures other than IA-32. Then, switching the pagetable base from A to B to A would possibly re-instate the mappings associated with A; if the kernel truly wanted to flush part or all of A, it would issue an explicit flush. Context switches could then be emulated efficiently on Itanium and other architectures supporting multiple contexts. This would not only to reduce the number of TLB misses, but also allow efficiently switching between pre-translated instruction streams for different processes. The binary translator, IA-32 EL in this case, would also need to be modified to allow such

switching.

If modifying the guest kernel is not practical, then tricks could be used to simulate a similar architecture given the information available. For example, on switching back to A, one could validate the Itanium pagetable entries against the IA-32 pagetable entries, or validate a checksum of the IA-32 pagetables. Such approaches might be expensive in the worst case (there may be many megabytes of IA-32 pagetables), but if done intelligently, are still likely to result in a performance gain when compared to the cost of re-creating the translations lazily via page faults.

In addition to context switches, there are also a number of other cases in the Linux kernel where a full TLB flush is used gratuitously, when the actual intention is only to flush a small number of mappings. It may be that on real IA-32 hardware the full TLB flush is faster than flushing the individual pages; however on a system like MagiXen it is significantly more expensive. Once again, one could either modify the guest kernel, or use workarounds based on detecting changes to the pagetables to try to limit the pages that are flushed.

In summary, our results show that the frequency of IA-32 TLB flushes poses one of the most significant challenges to providing good system performance for IA-32 code executing on a non-IA-32 platform. While we have implemented MagiXen on Itanium, we believe that similar problems would arise on other host architectures. We plan to further investigate possible solutions in the future.

6 Related Work

As the many references throughout this paper indicate, there is much previous work in both virtualization and DBT. We combine the benefits and many of the challenges of both, in order to support guest virtual machines consisting of an extremely popular source ISA on a rapidly growing target ISA.

As previously mentioned, VMware combines virtualization with binary rewriting, which is a form of binary translation. In this case, the binary translation occurs from one ISA to a subset of the same ISA. MagiXen is more flexible in that it supports an entirely different ISA. The DELI [14] provides an API and service to clients which allows translation or emulation of heterogeneous binaries. Because of its flexibility (specifically the fact that the API can be inserted below the operating system) it potentially could be used as a translation layer between a VMM and multiple guests, though the available literature does not suggest or propose this. For MagiXen, support of multiple guests, some native and some non-native, is a core objective. QEMU [10] is positioned as a fast emulator with dynamic translation capability for multiple guest/host combinations, providing a system emulation mode. It is approaching the objectives and possibilities of MagiXen from a slightly different tack, but we believe the fact that is is hosted will limit it from achieving some of the consolidation and migration benefits of a VMM-based solution and, as previously noted, its current performance for translated user programs is crippled without a dynamic optimizer.

7 Conclusions

In this paper, we have considered the notion of redefining the ISA at the virtualization layer. Specifically, we have proposed integrating dynamic binary translation functionality into a full-function virtual machine monitor. We have discussed how our proposed approach can extend the benefits of current applications of virtualization, with respect to uses such as server consolidation and resource provisioning More importantly, our approach is also well-aligned with recent trends towards virtual appliances and heterogeneous hardware. To understand the implementation issues, we built *MagiXen*, a prototype implementation of a Xen virtual machine monitor with integrated binary translation.

Our prototype can indeed seamlessly run IA-32 virtual machines built for Xen on an Itanium platform. We have presented performance results for several typical benchmarks and discussed our experiences with building the prototype. Our analysis reveals that our design, using a black-box commercial-grade application-level binary translator, provides superior performance for user-level code but results in a number of issues which have considerable impact on the performance of system code and memory intensive workloads. As part of ongoing work, we are exploring additional optimizations, but recognize that a more fully integrated design would likely overcome many of these issues — at the cost of a large engineering investment.

Overall, our results show that such a cross-ISA virtual machine with integrated binary translator can be a viable solution for future enterprise systems. Indeed, as a larger and larger fraction of enterprises adopt virtualization, we expect approaches like ours — which go beyond the conventional virtualization benefits of isolation, partitioning, and encapsulation to include other features like dynamic interface transformation — will be an important part of future virtualization solutions.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. ASPLOS XII*, Oct. 2006.

[2] Advanced Micro Devices Inc. AMD unveils Torrenza innovation socket. `http://www.hpcwire.com/hpc/917955.html`.

[3] T. Agerwala. Computer architecture: Challenges and opportunities for the next decade.

[4] E. R. Altman, K. Ebcioglu, M. Gschwind, and S. Sathaye. Advances and future challenges in binary translation and optimization. *IEEE Proceedings*,

89(11):1710–1722, Nov. 2001.

[5] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI throttling. In *Proceedings of the Int'l Symp. Computer Architecture, IEEE CS Press*, pages 298–309, 2005.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent runtime optimization system. In *Proc. PLDI '00*, Jun. 2000.

[7] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proc. 36th International Conference on Microarchitecture (MICRO36)*, 2003.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th SOSP*, pages 164–177, Oct. 2003.

[9] BEA Systems Inc. With VMware's help, BEA ditches the operating system. `http://www.dabcc.com/article.aspx?id=3257`.

[10] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[11] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2), Mar/Apr 1998.

[12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. 2nd NSDI*, 2005.

[13] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003.

[14] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A new run-time control point. In *35th Annual International Symposium on Microarchitecture (MICRO 'O2*, Nov. 2002.

[15] L. DiDio and G. Hamilton. Virtualization, Part 1: Technology goes mainstream, nets corporations big TCO gains and fast ROI. *Yankee Group*, July 2006.

[16] F. E. Gillett and G. Shreck. Pragmatic approaches to server virtualization: Flexible manageability drives adoption as users work around obstacles. *Forrester Research*, June 19 2006.

[17] E. Grochowski et al. Best of both latency and throughput. In *Proc. Int'l Conf. Computer Design, IEEE CS Press*, pages 236–243, 2004.

[18] Hewlett-Packard Company. Achieving binary affinity for HP-UX 11i for the Intel Itanium Processor Family, 2003. `http://h71028.www7.hp.com/erc/downloads/5982-0483EN.pdf`.

[19] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. Int'l Symp. on Microarchitecture*, San Diego, CA, Dec. 2003.

[20] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. In *IEEE Computer*, 2005.

[21] D. J. Magenheimer and T. W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *Proc. 3rd Virtual Machine Research and Technology Symposium*, pages 73–82, 2004.

[22] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *1st ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.

[23] Open Source Development Labs Inc. reaim benchmark suite. `http://sourceforge.net/projects/re-aim-7`.

[24] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

[25] I. Pratt et al. Xen 3.0 and the art of virtualization. In *Proc of 2005 Ottawa Linux Symposium*, volume 2, pages 65–77, July 2005.

[26] J. Rattner. Multi-core to the masses.

[27] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proc. 9th USENIX Security Symposium*, Aug. 2000.

[28] M. Rosenblum. The impact of virtualization on computer architecture and operating systems. Keynote at *ASPLOS XII*, Oct. 2006.

[29] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. 5th OSDI*, 2002.

[30] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufman, 2005.

[31] S. Soltesz, H. Pötzl, M. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. To appear in *Proc. EuroSys2007*.

[32] Standards Performance Evaluation Corporation. SPEC CINT2000 benchmarks. `http://www.spec.org/cpu/CINT2000/`.

[33] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proc. 2003 USENIX Technical Conference*, Boston, MA, Jun. 2001.

[34] Sun Microsystems Inc. Solaris operating system runs Linux applications easily, 2004. `http://www.sun.com/2004-0803/feature/`.

[35] The Wine project. `http://www.winehq.com/`.

[36] Transitive Corporation. QuickTransit software. `http://www.transitive.com/`.

[37] E. Traut. Building the Virtual PC. *BYTE Magazine*, pages 51–52, Nov. 1997. http://www.byte.com/art/9711/sec4/art4.htm.

[38] R. Uhlig, G. Neiger, D. Rodgers, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Com-*

puter, 38(5):48–56, May 2005.

[39] VMware Inc. Introducing the virtual appliance marketplace. http://vam.vmware.com/.

[40] VMware Inc. VMware Vmotion: Live migration of virtual machines without system interruption. http://www.vmware.com/pdf/vmotion_datasheet.pdf.

[41] D. Wentzlaff and A. Agarwal. Constructing virtual architectures on a tiled processor. In *Proc. 2006 International Symposium on Code Generation and Optimization*, 2006.

[42] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th OSDI*, Boston, MA, Dec. 2002.