UNIVERSITY OF CALIFORNIA, SAN DIEGO

Scalable Virtual Machine Multiplexing

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Diwaker Gupta

2009

The dissertation of Diwaker Gupta is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2009

DEDICATION

To my parents, for everything.

EPIGRAPH

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन।
मा कर्मफलहेतुर्भूर् मा ते संगोऽस्त्वकर्मणि॥
—*श्रीमद् भगवद् गीता, २.४७*

Let your claim lie on action alone and never on the fruits;
you should never be a cause for the fruits of action;
let not your attachment be to inaction.
—*The Bhagavad Gita, 2.47*
(English translation by Dr. S. Sankaranarayan)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

When I was applying to graduate school, I had no intention of joining a PhD program, let alone completing a PhD. I have come a long way since and this journey would not have been possible without the support of family, friends, colleagues and several others. While I have tried my best to be thorough in my acknowledgments, let me apologize beforehand for any omissions — the fault is entirely mine.

The cornerstone of a fruitful Ph.D. is the relationship between the adviser and the advised. Beyond the technical output of a dissertation, I believe that the real lessons learnt in graduate school lie in the non-technical knowledge gained from this relationship. I am truly grateful that I could work with Professor Amin Vahdat for my dissertation, and I will be forever indebted to him for giving me this opportunity. Amin has been a source of inspiration for me, both professionally and personally. Whenever I felt lost or drifted and lost focus, he was there to nudge me in the right direction; whenever I was being unproductive or feeling demotivated, he was there to gently correct me and provide the required moral boost. Amin's keen intellect and wisdom are matched only by his humility and modesty. I continue to learn from him and benefit from his experiences in all areas, be it time management, analyzing and approaching a new problem, maintaining a good work–life balance, or managing people and priorities.

One of the things I admire most about Amin is his attitude towards his students. He approaches each student differently, depending on the individual — he does not prescribe to the one-size-fits-all approach. I distinctly recall my first Ph.D. review with him. The Ph.D. review is an opportunity for the adviser and the student to discuss each others' strengths and weaknesses. At the outset, Amin told me that I had many strengths but discussing them would not help me. Instead, he suggested, that we discuss my weaknesses. At this point, he proceeded to succinctly point out four specific areas he thought I could improve upon. At first I was a little disappointed. But in hindsight, I realize that it was probably the most productive, useful and influential meeting I've had with Amin in all these years. I only hope that I have been able to improve myself in those areas. Amin is a true visionary. He has the uncanny ability to crystallize grand project ideas and instantly fashion a "big picture" out of the vague, disconnected ideas that I would present him with. He has been an absolute pleasure to work with, giving me the space to explore tangential ideas when I wanted, while at the same time keeping me from wandering aimlessly for too long.

Of course, I am indebted to several other faculty members in the SysNet group as well. I would especially like to express my gratitude to the following faculty members: Alex

Snoeren, for one of the most enjoyable classes I took at UCSD and for teaching me how to analyze graphs; Stefan Savage, for teaching me how to identify exciting problems and how to convert radical, outrageous sounding ideas into feasible and news-worthy research projects; Geoff Voelker, for coming up with the wonderful names for my papers, for his warm and genteel temperament and for teaching me how to evaluate a system; George Varghese, for teaching me how to put ideas into practice and how to convert theory into applied research.

But graduate school would be impossible without the support of colleagues and friends. I'm thankful to my friends in the systems group at large, and my lab-mates in particular for all the good times in school — sessions of playing foosball and darts, latte breaks and most importantly, the hallway conversations. In particular, I want to thank Alvin AuYoung, John McCullough, Justin Ma, Marti Motoyama, Priya Mahadevan, Qing Zhang, Ryan Braud, and Yuvraj Agarwal for their camaraderie — it has been an absolute pleasure and an honor working with these fine folk. One colleague, however, has been particularly inspiring for me: I started working with Kashi Venkatesh Vishwanath as a young graduate student and during the course of the next few years I learnt tremendously from him, especially about working smarter, not harder and the importance of automation and systematic benchmarking.

The sysadmin staff for our group has been amazing and instrumental in my dissertation. Marvin McNett is always a wealth of information and it was reassuring to have him by my side, both as support staff and later as fellow grad student. Chris Edwards is an endless source of amazement to me — how could a single person know so much about so many different things is something I still struggle with. I shall always cherish the long discussions we had. I also want to thank the administrative staff, in particular Michelle Panik, Kim Greeves and Paul Terry. No Ph.D. in the department would ever complete without the tireless perseverance and vigilance of Julie Conner, so many thanks to her as well.

I next want to thank the Indian graduate student community in UCSD. I made some lifelong friends during my time here. I specially want to thank my first house mates from Apartment 6224 and members of the UCSDPanduz — you have given me some wonderful memories! My heartfelt gratitude to my friends out of town who provided me with the much needed support and encouragement from hundreds and thousands of miles away — Asim Shankar, Priyendra Singh Deshwal, Abhaya Agarwal, Vibhore Vardhan and Ravi Shankar Srivastava warrant a special mention.

My acknowledgments would be incomplete without a mention of the free and open source software community. I'm a passionate user and advocate, and a modest contributor of open source software. This dissertation would not have been possible without open source: from the Xen virtual machine monitor on which my entire dissertation is built, to the tools I use on a daily basis for conducting and publishing my research. So thank you, all the tireless, unsung heroes whose voluntary efforts have given millions quality software for free, and more importantly, an opportunity to understand and learn the way these applications work.

And finally, I would like to thank my family. I am what I am today because of my parents. They are the pillars of support and inspiration in my life. I thank them for simply being them, for their unwavering trust and confidence, and for their guidance and wisdom. I only wish I can bring as much pride to them as they bring to me. I thank my sister for being the perfect elder sister. Her constant love and support are hugely motivating and I thank her for giving me perspective on so many things in life. Last, but not the least, I want to thank my wife, Surabhi, for making my life a lot more adventurous and exciting than it used to be. She has been a wonderful listener, and her patience and encouragement saw me through the most frustrating of times.

I always believed that a Ph.D. is more about the journey than the reaching the destination. Having gone through the process, the belief has turned into a conviction. This journey could not have been possible without the tremendous support of friends, family and colleagues and I am grateful to them for this opportunity.

Chapters 1, 4 and 5, in part, are reprints of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2006. Gupta, Diwaker; Yocum, Kenneth; McNett, Marvin; Snoeren, Alex C.; Vahdat, Amin; Voelker, Geoffrey M. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 3, 5 and 6, in part, are reprints of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2008. Gupta, Diwaker; Vishwanath, Kashi V.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 7 and 8, in part, are reprints of the material as it appears in Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI) 2008. Gupta, Diwaker; Lee, Sangmin; Vrable, Michael; Savage, Stefan; Snoeren, Alex C.; Varghese, George; Voelker, Geoffrey M.; Vahdat, Amin. The dissertation author is the primary investigator and author of this paper.

VITA

2003	B.Tech. in Computer Science and Engineering, Indian Institute of Technology, Kanpur, India
2006	M.S. in Computer Science, University of California, San Diego
2009	Ph.D. in Computer Science, University of California, San Diego

PUBLICATIONS

“Difference Engine: Harnessing Memory Redundancy in Virtual Machines.” Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker and Amin Vahdat. *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (**OSDI**), 2008.

“DieCast: Testing Distributed Systems with an Accurate Scale Model.” Diwaker Gupta, Kashi V. Vishwanath and Amin Vahdat. *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (**NSDI**), 2008.

“Usher: An Extensible Framework for Managing Clusters of Virtual Machines.” Marvin McNett, Diwaker Gupta, Amin Vahdat and Geoffrey M. Voelker. *Proceedings of the 21st Large Installation Systems Administration Conference* (**LISA**), 2007.

“Enforcing Performance Isolation Across Virtual Machines.” Diwaker Gupta, Ludmila Cherkasova, Rob Gardner and Amin Vahdat. *Proceedings of the 7th ACM/IFIP/USENIX Middleware Conference* (**MIDDLEWARE**), 2006.

“To Infinity and Beyond: Time-Warped Network Emulation.” Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat and Geoffrey M. Voelker. *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation* (**NSDI**), 2006.

“Routing in an Internet-Scale Network Emulator.” Jay Chen, Diwaker Gupta, Kashi V. Vishwanath, Alex C. Snoeren and Amin Vahdat. *Proceedings of the 12th ACM/IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (**MASCOTS**), 2004.

ABSTRACT OF THE DISSERTATION

Scalable Virtual Machine Multiplexing

by

Diwaker Gupta

Doctor of Philosophy in Computer Science

University of California, San Diego, 2009

Professor Amin Vahdat, Chair

A *virtual machine* (VM) is a software abstraction of a real, physical machine. Virtualization has been around for almost 50 years, beginning with IBM's pioneering work in the 1960s. However, recent years have seen a significant surge in the interest and use of virtualization, driven by better hardware support, increasing power costs and low resource utilizations. Server consolidation is by far the most common application for virtualization — by aggregating multiple services on a single physical machine, organizations can reduce costs and increase the utilization of their infrastructure. While server consolidation remains a powerful driving force, virtualization is now becoming even more compelling for the innovative applications it enables, such as to support legacy software, for disaster recovery and backup, and for intrusion detection and malware analysis, to name a few.

The key premise of this dissertation is that the ability to efficiently multiplex virtual machines is critical to realizing the benefits of virtualization, not just for existing applications such as server consolidation, but also in enabling some fundamentally new applications. An inevitable consequence of conventional mechanisms for multiplexing is resource partitioning: individual VMs can only use a fraction of the actual physical resources. This partitioning limits not only the resources available for a single VM, but also the total number of VMs that the hardware can support. Here we hit a fundamental barrier — *the aggregate resources* *available to the VMs are bounded by the capacity of the underlying hardware*. This dissertation describes mechanisms to work around this barrier to increase the *perceived* resource capacity of individual VMs, as well as to increase the total number of VMs that can be created.

First, we present "time dilation," a technique that allows the perceived aggregate capacity of the VMs to exceed the capacity of the underlying hardware. Time dilation also enables many interesting experiments, such as predicting application behavior and protocol performance in resource-rich environments. Next, we use time dilation to build a framework and methodology called DieCast for accurate testing of large systems using a much smaller infrastructure. Finally, we present Difference Engine, a system that exploits fine-grained similarities in memory among VMs to extract twice as much memory savings as the current state of the art, thus freeing memory for additional VMs. Together, our contributions make significant advances in making virtual machine multiplexing more scalable.

Chapter 1

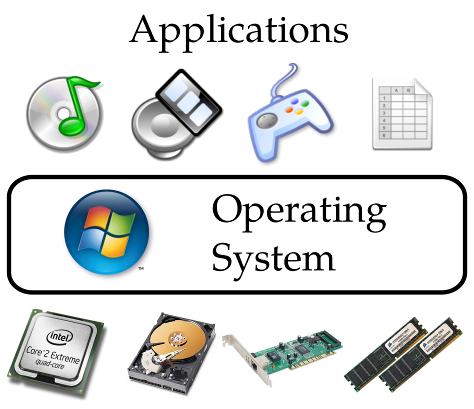
Introduction

It is not often that an old idea, one that has been around for more than half a century, is suddenly able to generate the kind of interest and excitement that *virtualization* has done in recent years. Virtualization refers to the methodology of partitioning a single, physical machine into independent, isolated containers, each of which can host its own operating system and application stack. A virtual machine environment differs significantly from the conventional operating system model for managing resources on a physical machine.

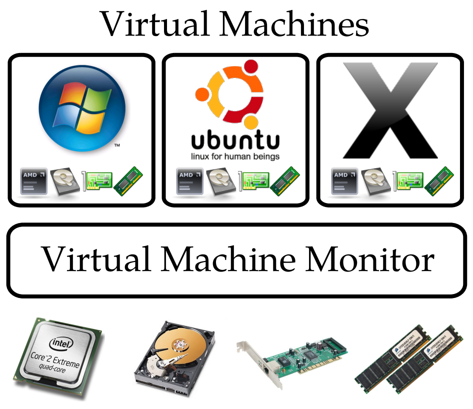
Figure 1.1a shows a regular physical machine where the operating system (OS) is responsible for managing the various system resources such as CPU and main memory among several applications. In a virtualized environment (Figure 1.1b), instead of the OS managing the hardware resources, there is a thin software layer called the *virtual machine monitor* (VMM) that is responsible for managing the system resources among several virtual machines (VMs), sometimes also referred to as *domains*. The VMM is sometimes also referred to as the *hypervisor* — we use the two terms interchangeably in this dissertation.

Popek and Goldberg [88] define a virtual machine to be an "efficient, isolated duplicate of the real machine." Thus, a virtual machine is essentially a software abstraction of a real, physical machine. Each VM can have its own OS and application stack. Typically, this is completely transparent to software running inside the virtual machine — for all practical purposes, the software believes that it is actually running on real hardware. The operating system running in a VM is called a *guest OS*. The hardware and software techniques and the methodology used to support the VM abstraction are collectively referred to as virtualization.

The history of virtualization — its origins and subsequent obsolescence for nearly three decades, followed by the recent resurgence — is instructive in that it demonstrates how



(a) The operating system is conventionally responsible for managing system resources among several applications.



(b) The virtual machine monitor multiplexes the physical resources among several virtual machines. The operating system within each virtual machine believes it is running on real hardware.

Figure 1.1: A virtual machine environment vs. a regular operating system.

technology that is ahead of its time is sometimes abandoned, and how good ideas eventually are recycled. IBM did much of the pioneering work in virtualization in the 1960s when computers were huge, expensive and inaccessible to but a select few. As a result, there was significant interest in techniques that would allow multiple users to simultaneously access a computer. Time-sharing and multiprogramming were borne out of the initial efforts to support multi-user environments.

Virtualization emerged as a natural extension to these ideas, since a virtual machine could cleanly encapsulate the entire operating system and application stack, completely isolating users from one another. In a regular operating system, users often share the file system and software stack along with all the installed applications and libraries. Furthermore, resource accounting and allocation typically happens at process granularity in such systems. Virtual machines simplify the allocation and accounting, especially if each user is using several processes. Each user can also set up his or her own distinct software environment — right from the underlying operating system up to the applications — independent of other users on the machine. Thus, VMs offered a much cleaner and more complete abstraction than simple process-based address-space isolation.

Another early motivation for virtualization was to provide portability and backwards compatibility. At that time, every new system was different than previous systems both in terms of the low-level hardware interface, as well as the system software. As a result, software had to be rewritten from scratch for each different system. IBM pioneered the use of virtualization to support both forward and backward compatibility by standardizing the instruction set [43]. We take standardized architectures granted for today, but back then this was a revolutionary concept. By using a virtual machine monitor, they could guarantee that the operating system would always see a consistent hardware interface, regardless of the actual hardware running underneath. A side-effect of this instruction-level standardization was that it became possible for other hardware vendors to build IBM-compatible hardware. IBM's monopoly established its hardware platform as the *de facto* architecture for which software was written. As a result, new entrants in to the computer hardware market had incentive to provide compatibility with software written for IBM hardware by using an appropriate virtualization layer [1]. These virtualization layers often exported a different hardware interface than the physical hardware.

As personal computers and commodity servers replaced mainframes, clusters of inexpensive, off-the-shelf equipment emerged as the preferred model for computing. As opposed to a centralized mainframe, compute resources were distributed across a large number of machines. Complicated mechanisms such as virtualization were an overkill for low-end machines like the initial PCs. At the same time, the emergence of portable, commodity OSes like Unix and Windows provided a common platform for software developers, further obviating the need for virtualization to provide hardware compatibility. As a result, interest in virtualization gradually faded and the idea was largely abandoned for the next two decades.

However, by the late 90s, personal computers had become powerful enough that commodity operating systems were not able to effectively utilize resources such as multiple processors and large amounts of memory. This led to research efforts such as Disco [27] in 1997 that explored the use of virtualization as a means of enabling commodity OSes to use the hardware more effectively without having to significantly modify or rewrite the operating system. Over the past decade, interest in virtualization has continued to increase steadily. In particular, the last few years have seen a significant surge in the interest and use of virtualization, both in the industry and academia. Virtualization was a $5.5 billion industry in 2007 and is expected to grow to $11 billion by 2011 [2]. Another study estimates that by 2009, two-thirds of all IT departments will virtualize almost half of their servers [57].

While there are many factors contributing to this renewed interest, they can be summarized thus: virtualization enables more *efficient* use of existing resources. In other words, organizations see virtualization as a way of getting more value out of their infrastructure. The overarching theme of this dissertation is in the same spirit: to extract more value and utility from a given infrastructure using virtualization. Let us take a closer look at the two dominant driving forces behind virtualization.

The first is based around *total cost of ownership* (TCO). While the infrastructure costs of a datacenter have remained largely stable, the administrative costs and the power and cooling costs have increased at an alarming rate and continue to do so [3, 77, 91]. Further, anecdotal evidence suggests that typical datacenter servers are severely under-utilized, typically running at 5–15% utilization [4] because of the need to over-provision for peak levels of demand, because fault isolation mandates that individual services run on individual machines, and because many services often run best on a particular operating system configuration (precluding service sharing for a particular configuration). Hence, enterprises have significant incentive to consolidate their servers using virtualization: by aggregating multiple services on fewer physical machines, organizations can reduce costs and increase resource utilization levels.

The other and perhaps more important reason why virtualization is increasingly more compelling is its versatility: virtualization has enabled novel applications in many different areas. For instance, since the VMM can export different hardware interfaces to the guest OS, virtualization can support legacy software and hardware, even if the real hardware does not exist or is unavailable. Virtual machines provide a secure, isolated container to run an operating system. The powerful system-level monitoring and introspection offered by VMs is useful for many applications such as intrusion detection [47] and malware detection [103, 69]. VMs also provide fine-grained (instruction-level) logging, useful for debugging operating systems [70]. Finally, virtual machines can be created on demand and migrated from one physical machine to another. This ability gives organizations tremendous flexibility in provisioning resources, and allows them to rapidly adapt to failures [84] and changing workloads. Section 2.2 offers a more comprehensive list of the various benefits of virtualization. Virtualization thus presents unique opportunities, both as a vehicle for academic research, and also for commercial products.

This dissertation makes contributions in both of these domains. The key premise is that the ability to efficiently multiplex virtual machines is critical to realizing the benefits of virtualization. An obvious consequence of conventional mechanisms for multiplexing virtual machines is *resource partitioning*: virtual machines hosted on the same physical machine will share the underlying physical resources, and hence, individual VMs can only use a fraction of the actual resources. Resource partitioning limits not only the resources available for a single

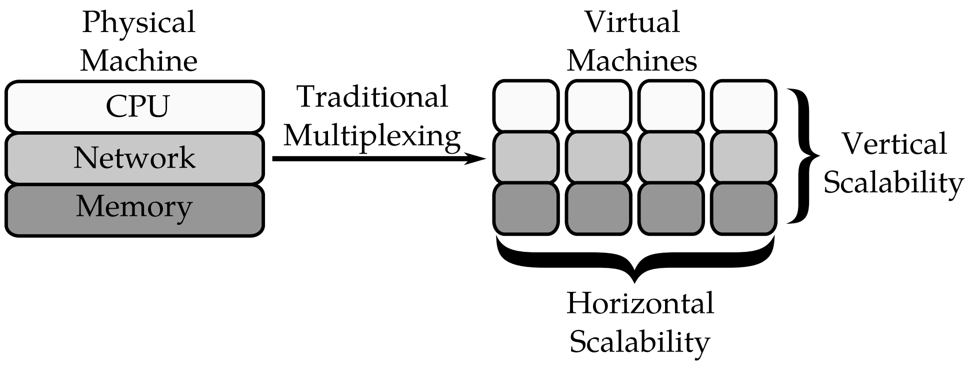


Figure 1.2: With traditional multiplexing, on a single physical machine hosting four virtual machines, each VM gets a fraction of the underlying physical resources.

VM, but also the total number of VMs that the hardware can support (see Figure 1.2). We refer to the former problem as *vertical scalability* and the latter as *horizontal scalability*. Note that these limits are not independent — increasing the number of VMs would reduce the resources available to any single VM. Together, they represent a *fundamental barrier* — the aggregate resources available to the VMs on a single physical machine are bounded by the capacity of the underlying hardware. Similarly, the utility of a given set of virtual machines is limited by the total hardware capacity of the underlying physical infrastructure.

We show that both these challenges can be addressed by reconsidering conventional mechanisms for resource multiplexing. This dissertation claims that alternate mechanisms for resource multiplexing exist. Further, these mechanisms not only support existing applications of virtualization in areas such as server consolidation, but also enable several unique and important applications that would not have been possible otherwise.

Addressing horizontal scalability means increasing the number of VMs that a single physical machine can support. A higher multiplexing factor allows for more aggressive server consolidation, for instance, translating to reduced costs and increased utilization. However, it is unclear what it means to address vertical scalability in this context. We cannot possibly create more physical resources out of void, so clearly the real resource capacity of a VM will *always* be bounded. However, what can be manipulated is the *perceived* resource capacity of individual VMs. This dissertation presents mechanisms to allow the perceived resource capacity of VMs to exceed that of the underlying hardware. We argue that this ability enables some unique applications that would have been otherwise impossible.

We next present two concrete problems that can benefit from the capability to go beyond the hardware capacity. Both these problems are particularly relevant and timely in today's world of increasingly large and complex systems that we interact with on a daily basis. The Internet is an obvious example, but air-traffic control systems, stock exchanges, cellular and telephone networks, cable and satellite television are all examples of such large, distributed systems. Because of the increasing dependence of our socio-economic infrastructure on such systems, our ability to effectively test and accurately evaluate them — especially at scale — is paramount. Unfortunately, as the following examples show, all too often we are limited in our testing and empirical evaluation by the capacity of the underlying hardware.

1.1 Scalable Network Emulation

Network emulation is a popular and widely used technique for testing systems, predicting system behavior and exploring application performance [102]. However, researchers are forced to work within the constraints of the capacity of the underlying hardware, for example, the bandwidth of the underlying network. While more powerful machines and higher capacity networks may exist, they will probably be accessible to a select few, and, even then, availability might be constrained geographically as well as temporally. Further, for certain regimes, the physical hardware might simply not exist (such as terabit wide-area links). Software-based simulators also make it possible to explore arbitrary resource regimes. However, a complete software simulation lacks realism — ideally we would like to use real hardware with unmodified operating systems and applications. The ability to experiment beyond the physical capacity of underlying hardware presents a number of interesting applications. Consider the following scenarios:

- **Emerging I/O technologies.** Imagine a complex cluster-based service interconnected by 100-Mbps and 1-Gbps Ethernet switches. The system developers suspect overall service performance is limited by network performance. However, upgrading to 10-GigE switches and interfaces involves substantial expense and overhead. The developers desire a low-cost mechanism for determining the potential benefits of higher-performance network interconnects before committing to the upgrade.

- **Scalable network emulation.** Today large ISPs cannot evaluate the effects of modifications to their topology or traffic patterns outside of complex and high-level simulations. While they would like to evaluate internal network behavior driven by realistic traffic traces, this often requires accurate emulation of terabits per second of bisection bandwidth.

- **High bandwidth-delay networking.** We have recently seen the emergence of computational grids [20, 51] inter-connected by high-speed and high-latency wide-area interconnects. For instance, 10-Gbps links with 100–200 ms round-trip times are currently feasible. Unfortunately, existing transport protocols, such as TCP, deliver limited throughput to flows sharing such a link. A number of research efforts have proposed novel protocols for high bandwidth-delay-product settings [65, 74, 113, 112, 63, 52, 67]. However, evaluation of the benefits of such efforts is typically relegated to simulation or to those with access to expensive wide-area links. But even with access to such links, it is not possible to experiment with "what if" scenarios.

1.2 Large-Scale Testing

Today, more and more services are being delivered by complex systems consisting of large ensembles of machines spread across multiple physical networks and geographic regions. Economies of scale, incremental scalability, and good fault-isolation properties have made clusters the preferred architecture for building planetary-scale services. A single logical request may touch dozens of machines on multiple networks, all providing instances of services transparently replicated across multiple machines. Services consisting of hundreds of thousands of machines now exist [33, 95].

Economic considerations have pushed service providers to a regime where individual service machines must be made from commodity components—saving an extra $500 per node in a 100,000-node service is critical. Similarly, nodes run commodity operating systems, with only moderate levels of reliability, and custom-written applications that are often rushed to production because of the pressures of "Internet Time." In this environment, failure is common [85] and it becomes the responsibility of higher-level software architectures, usually employing custom monitoring infrastructures and significant service and data replication, to mask individual, correlated, and cascading failures from end clients [97].

One of the primary challenges facing designers of modern network services is testing their dynamically evolving system architectures. In addition to the sheer scale of the target systems, challenges include: heterogeneous hardware and software, dynamically changing request patterns, complex component interactions, failure conditions that only manifest under high load [75], the effects of correlated failures [59], and bottlenecks arising from complex network topologies. Before upgrading any aspect of a networked service—the load balancing/replication scheme, individual software components, the network topology—architects would ideally create an exact copy of the system, modify the single component to be upgraded, and then subject the entire system to both historical and worst-case workloads. Such testing must include subjecting the system to a variety of controlled failure and attack scenarios since problems with a particular upgrade will often only be revealed under certain specific conditions.

Creating an exact copy of a modern networked service for testing is often technically challenging and economically infeasible. The architecture of many large-scale networked services can be characterized as "controlled chaos," where it is often impossible to know exactly what the hardware, software, and network topology of the system looks like at any given time. Even when the precise hardware, software and network configuration of the system is known, the resources to replicate the production environment might simply be unavailable, particularly for large services. And yet, reliable, low-overhead, and economically feasible testing of network services remains critical to delivering robust higher-level services. As one motivating example, consider that the Nikkei Stock Exchange recently shut down for a day [22] while the New York Stock Exchange indicated an inaccurate precipitous price drop (dropping 200 points almost instantly) [9] as a result of, in both cases, unusually high trading volumes. Today, testing Internet services is relegated to small-scale deployments on hardware, software, and interconnects that only approximate the target architecture.

Instead, imagine if we had a framework to accurately predict the behavior of modern network services while *employing an order of magnitude less hardware*. For example, consider a service consisting of 10,000 heterogeneous machines, 100 switches, and hundreds of individual software configurations. Consider the benefits of configuring a small number of machines (e.g., 100–1000 depending on service characteristics) to emulate the original configuration as closely as possible. We could then subject this test infrastructure to the same workload and failure conditions of the original service. The performance and failure response of the test system should closely approximate the real behavior of the target system. Of course, these goals are infeasible without giving something up: if it were possible to capture the complex behavior and overall performance of a 10,000 node system on 1,000 nodes, then the original system should likely run on 1,000 nodes.

The ability to increase the perceived resource capacity of VMs, allowing the aggregate capacity to exceed that of the underlying hardware, makes such a framework possible. Mechanisms to increase vertical scalability are therefore critical for such applications. But note that such a framework would benefit from horizontal scalability as well — packing more VMs on fewer physical machines will reduce the hardware required to replicate and test a given system even further. As we mentioned earlier, horizontal scalability also supports existing applications such as server consolidation.

1.3 Challenges

Standard mechanisms for resource multiplexing in VMMs provide neither horizontal nor vertical scalability. For instance, CPU and network are typically time-shared among VMs while main memory is often simply statically partitioned. Scaling a system *both* horizontally and vertically implies that the system somehow sees more resources than the actual physical resources available. Clearly, we must give something up in order to do this. In this section we highlight the main challenges and trade-offs in achieving vertical and horizontal scalability.

1.3.1 Vertical Scalability

Since we do not have control over the availability of actual physical resources, our approach to address vertical scalability is to instead simply convince the operating system (and, hence, all applications) that it has more resources than it actually does. As this dissertation will demonstrate, if implemented properly, this perception alone is extremely powerful even though the real resource availability remains unchanged. To be useful, this perception must be:

- **transparent**, meaning that existing operating systems and applications should not require any modifications. Transparency is desirable because we want to our approach to be broadly applicable: requiring operating system support not only involves significant implementation overhead to support multiple platforms, but also assumes that operating systems can be trusted and users are not malicious.

- **pervasive**, meaning that the operating system and applications should not be able to detect that the resource capacity they perceive does not reflect reality — the resources must appear and behave exactly as real resources. A pervasive implementation is required to present a consistent view of the world — if different components in a system perceive different realities, it can lead to unexpected behavior.

Next, in order to validate such an approach, we need to compare an existing system that matches the perceived resource capacity of *some* test system. That is, we must compare two systems such that the real resource capacity of one matches the perceived resource capacity of the other. Appropriately configuring systems for such *resource equivalence* requires fine-grained control over the allocation of various system resources such as CPU and network bandwidth. Such control is particularly challenging for disk I/O: no good mechanisms exist to control the quality of service for disk I/O (such as throughput and latency) even in regular operating systems, let alone virtual machine monitors.

Thus the key challenges for achieving vertical scalability are implementing a broadly applicable mechanism for creating a high-fidelity alternate reality for VMs where the perceived resource capacity differs from the real resource capacity, and providing fine-grained, independent knobs for controlling the actual physical resources available to each virtual machine.

1.3.2 Horizontal Scalability

We noted earlier that virtual machines are particularly attractive for server consolidation. Their strong resource and fault isolation guarantees allow multiplexing of hardware among individual services, each configured with a custom operating system. One of the promises of virtual machine technology for server consolidation is to run many such services on a single physical machine while still allowing independent configuration and failure isolation. This is precisely the notion of horizontal scalability. The first step in increasing horizontal scalability is to identify the primary bottleneck in creating more VMs per physical machine. Since each additional VM consumes some system resources, the primary bottleneck is the system resource that is exhausted most quickly.

First, note that the CPU requirements of applications (and frequently, VMs, since a common model is to dedicate one VM per application) is often bursty [39, 41]. Such bursty usage patterns make statistical multiplexing particularly attractive: systems can be provisioned for average load since the likelihood of all co-located VMs hitting peak load are quite small.

However, while physical CPUs are frequently amenable to such multiplexing, main memory is not. For instance, while many services run comfortably with 1 GB of main memory, a multiplexing degree of 10 would require that each VM running on a host be allocated just 100 MB of main memory. Increasing a machine's physical memory is often both difficult and undesirable. Incremental upgrades in memory capacity are subject to both the availability of extra slots on the motherboard and the ability of the memory chipset to support higher-capacity modules. Moreover such upgrades often involve replacing—as opposed to just adding—memory chips. Not only is high-density memory expensive, it also consumes significant power. Looking forward, as high density multi-core processors become the norm, the bottleneck for VM multiplexing will increasingly be the memory, not the CPU. Finally, applications and operating systems are becoming more and more resource-intensive. As a result, commodity operating systems require significant physical memory to avoid frequent paging.

Not surprisingly, researchers and commercial VM software vendors have focused significant attention on decreasing the memory requirements for virtual machines. Notably, the VMware ESX server implements content-based page sharing, which has been shown to reduce the memory footprint of multiple, homogeneous virtual machines by 10–40% [110]. We find that these values depend greatly on the operating system and configuration of the guest VMs. We are not aware of any previously published sharing figures for mixed-OS ESX deployments. Our evaluation indicates, however, that the benefits of ESX-style page sharing decrease as the heterogeneity of the guest VMs increases, due in large part to the fact that full-page sharing requires the candidate pages to be *identical*. We claim that there are significant additional benefits from sharing at a sub-page granularity, i.e., there are many pages that are *nearly* identical. The challenge then is to efficiently find such similar pages and to coalesce them into a smaller memory footprint without degrading application performance.

1.4 Contributions

Scalable multiplexing poses several challenges, and, at the same time, presents new opportunities for applying virtualization in scalable network emulation and large-scale testing. This dissertation explores alternative approaches for resource multiplexing in virtual machines and the applications they enable. We claim that 1) we can increase the perceived resource capacity of VMs (to address vertical scalability) by giving up time itself, and that 2) more efficient memory management mechanisms for VMMs exist, and that these mechanisms can increase the number of VMs that can be supported on a single physical machine (for horizontal scalability). Increasing the perceived resource capacity of a given infrastructure allows us to accurately replicate and test much larger systems, while packing more VMs on a single physical machine enables aggressive server consolidation. We validate our claims by providing comprehensive implementations for a popular open-source virtual machine monitor. We further demonstrate how our mechanisms can be used in practice to attack some of the concrete problems we introduced earlier in the chapter. Specifically, this dissertation makes the following contributions:

1.4.1 Time Dilation

First, we develop a technique called time dilation to allow the perceived aggregate capacity of a system to exceed the capacity of the underlying hardware. Our implementation supports unmodified applications running in commodity operating systems and stock hardware. The interposition between a VM and the VMM provides the additional potential to independently dilate time for each hosted virtual machine.

- **Time-dilated virtual machines.** We show how to completely encapsulate a host running a commodity operating system in an arbitrarily dilated time frame within a VM. We allow processing power and I/O performance to be scaled independently (e.g., to hold processing power constant while scaling I/O performance by a factor of 10, or vice versa).
- **Accurate network dilation.** We perform a detailed comparison of TCP's complex end-to-end protocol behavior—in isolation, under loss, and with competing flows—under dilated and real time frames. We find that both the micro and macro behavior of the system are indistinguishable under dilation. To demonstrate our ability to predict the performance of future hardware scenarios, we show that the time-dilated performance of an appropriately dilated six-year old machine with 100-Mbps Ethernet is indistinguishable from a modern machine with Gigabit Ethernet.
- **End-to-end experimentation.** We demonstrate the utility of time dilation by experimenting with a content delivery overlay service. In particular, we explore the impact of high-bandwidth network topologies on the performance of BitTorrent [7], emulating multi-gigabit bisection bandwidths using a traffic shaper whose physical capacity is limited to 1 Gbps.

1.4.2 DieCast

Next, we leverage time dilation to build DieCast, a complete environment for constructing accurate models of network services. DieCast is not just a framework, but also a methodology for large scale testing of systems using a much smaller infrastructure. Our goals with DieCast are to enable accurate testing of large systems at scale while preserving realism.

We run the actual operating systems and application software of some target environment on a fraction of the hardware in that environment. To support complete system evaluations, DieCast provides mechanisms to independently scale individual resources such as CPU, network and disk in a dilated time frame. In particular, we integrate a full disk simulator into the virtual machine monitor (VMM) to consider a range of possible disk architectures. Finally, we conduct a detailed system evaluation, quantifying DieCast's accuracy for a range of services, including a commercial storage system.

1.4.3 Difference Engine

Main memory increasingly seems to be the primary bottleneck limiting the number of virtual machines that can be supported by a single physical machine. To address this, we present Difference Engine, an extension to the Xen virtual machine monitor that not only shares identical pages, but also supports sub-page sharing and in-memory compression of infrequently accessed pages. Among the set of similar pages, we are able to store multiple pages as *patches* relative to a single baseline page. We view these sub-page patches as a form of compression — one in which existing pages act as a pre-existing dictionary. Traditional stream-based compression algorithms typically do not have sufficient "look-ahead" to find commonality across a large number of pages or across large chunks of content, but they can exploit commonality within a local region, such as a single memory page. Thus, we employ traditional compression for those pages that are are not good candidates for patching and also unlikely to be accessed in the near future. We show that an efficient implementation of compression nicely complements whole-page sharing and patching.

Our results show that Difference Engine can reduce the memory footprint of homogeneous workloads by up to 90%, a significant improvement over previously published systems [110]. For a heterogeneous setup (different operating systems hosting different applications), we can reduce memory usage by nearly 70%. In a head-to-head comparison against VMware's ESX server running the same workloads, Difference Engine delivers a factor of 1.5 more memory savings for a homogeneous workload and a factor of 1.6-2.5 more memory savings for heterogeneous workloads. Critically, we demonstrate that these benefits can be obtained without negatively impacting application performance. In fact, we show that Difference Engine can leverage this improved memory efficiency to increase aggregate system performance by utilizing the free memory to create additional virtual machines in support of a target workload. For instance, one can improve the aggregate throughput available from multiplexing virtual machines running web services onto a single physical machine.

In summary, our contributions enable us to push beyond hardware limits by two orders of magnitude for scalable network emulation, while preserving realism and accuracy. Similarly, our techniques can be used to test a large scale system by using an order of magnitude smaller infrastructure. Finally, we demonstrate a factor of two improvement in horizontal scalability by making more efficient use of system memory. This work makes virtual machine multiplexing significantly more scalable compared to the current state-of-the-art.

1.5 Organization

Chapter 2 covers the requisite background and concepts that are frequently referred to in this dissertation. It also discusses the related work.

We introduce the DieCast framework in Chapter 3 with an overview of the DieCast architecture. The chapter concludes by highlighting the requirement of time dilation and accurate resource scaling to make DieCast possible. In Chapter 4, we describe time dilation in detail. We discuss general implementation strategies and then describe our implementation of time dilation in Xen. We then present a comprehensive validation and evaluation of time dilation. While time dilation uniformly scales all temporal resources, DieCast additionally requires mechanisms to independently control CPU, disk and network resources in a dilated time frame. These are the subject matter of Chapter 5. We present the implementation and microbenchmarks for each of the mechanisms. Chapter 6 evaluates DieCast across four representative networked services, including a commercial file system.

Chapter 7 discusses the design and implementation of Difference Engine, our framework for extracting memory savings across VMs by considering sub-page granularity redundancies. We evaluate Difference Engine using microbenchmarks and a comprehensive suite of workloads, including head-to-head comparisons against the VMware ESX server in Chapter 8.

Finally Chapter 9 presents some future work and concludes with a summary of the dissertation.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2006. Gupta, Diwaker; Yocum, Kenneth; McNett, Marvin; Snoeren, Alex C.; Vahdat, Amin; Voelker, Geoffrey M. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of USENIX

Symposium on Networked Systems Design and Implementation (NSDI) 2008. Gupta, Diwaker; Vishwanath, Kashi V.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI) 2008. Gupta, Diwaker; Lee, Sangmin; Vrable, Michael; Savage, Stefan; Snoeren, Alex C.; Varghese, George; Voelker, Geoffrey M.; Vahdat, Amin. The dissertation author is the primary investigator and author of this paper.

Chapter 2

Background and Related Work

Virtualization and virtual machines are much overloaded terms. For instance, many dynamic programming languages such as Java, Python and Ruby have run-time environments that are called virtual machines. We begin this chapter by clarifying the terminology and definitions of various virtualization concepts in the context of this dissertation. We also discuss some of the benefits and applications of virtualization.

While the ideas in this dissertation are applicable to virtual machine monitors in general, most of our implementation is built on top of Xen [35], an open-source virtual machine monitor. We give a brief overview of Xen in this chapter.

Finally, our work builds upon previous efforts in a number of areas, particular network emulation, large-scale testing, memory management and virtual machine multiplexing. We discuss the salient related work in each of these areas below.

2.1 Virtualization Concepts

There are several approaches to virtualization, and hence, several different ways of classifying virtualization platforms. A common classification in the literature, due to Goldberg [56], is to distinguish between *Type I* and *Type II* VMMs. Figure 2.1 depicts the differences between these two types of VMMs. A Type I VMM, also known as a *bare-metal hypervisor*, executes directly on the physical machine. Thus, in this model, the hypervisor is responsible for managing the low level system resources across virtual machines. Xen [35] and VMware ESX server fall in this category. A Type II VMM, also known as a *hosted hypervisor*, differs in that it needs a host operating system underneath – essentially it runs like any other user-

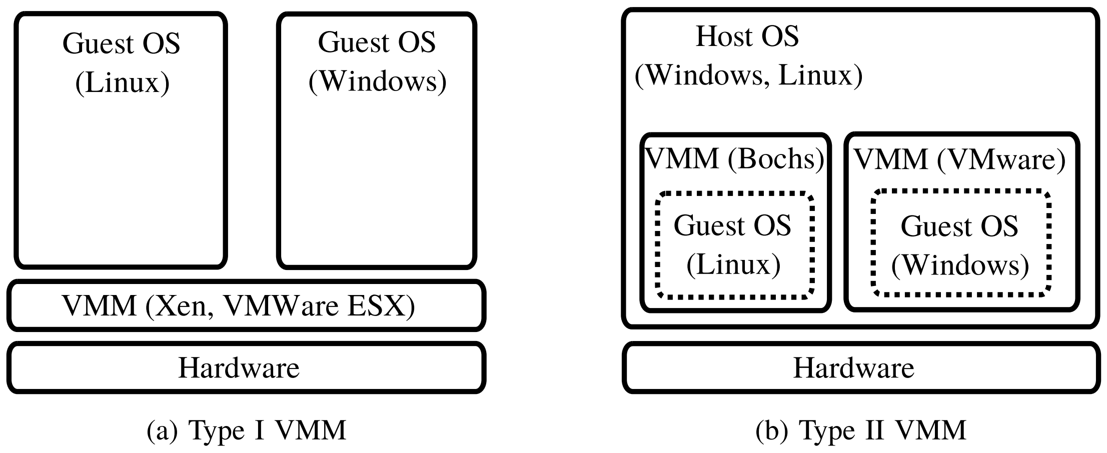


Figure 2.1: VMM taxonomy: Type I VMMs execute directly on the hardware, while Type II VMMs need a host OS to run.

space application. Therefore, a Type II VMM depends on the underlying host OS for resource management. VMMs such as Bochs [73], VMware Workstation Edition, and UML [46] would fall in this category. Unless otherwise stated, we assume a Type I hypervisor in this dissertation.

One of the primary responsibilities of the VMM is to manage system resources among VMs. Much like an operating system, the primary resources a VMM manages are CPU cycles, main memory, network I/O and disk I/O. It is fair to say among these, mechanisms for CPU management for VMs are the most sophisticated, in part due to the rich literature on process scheduling in operating systems. Mechanisms for managing memory and network I/O are simplistic in comparison. Main memory is typically statically partitioned among VMs. Most VMMs provide mechanisms to either share the network bandwidth among VMs equally, or to implement simple quality-of-service policies. Simple best-effort, fair-share scheduling represents the state of the art in disk I/O management.

VMM CPU schedulers can be characterized along several dimensions. For instance, *proportional share* (PS) schedulers allocate CPU according to the relative weights assigned to the virtual machines. In contrast, some schedulers might allow specifying *reservations*: absolute upper limits on the amount of the CPU that a virtual machine may consume. While PS schedulers are simpler to implement and easier to reason about, performance critical applications often have to meet service level agreements and would benefit from guaranteed resource reservations.

Schedulers can also be classified as supporting *work-conserving* and/or *non-work-*

conserving modes. In work-conserving mode, the shares or reservations are meaningful only for non-idle VMs. That is, idle VMs have no claim on the unused CPU cycles in their *quota* and the slack CPU can be allocated to any other eligible VM. Thus the CPU is idle if and only if there are no runnable VMs in the system. For example, in the case of two VMs with equal weights, if one of the VMs is idle, the other VM can consume 100% of the CPU. The exact opposite is true for the non-work-conserving mode. Any unused CPU cycles allocated to VMs get wasted — they are not accessible to any other VM. Considering the previous example of two VMs with equal weights, each VM can individually consume a maximum of 50% of the CPU at any point, even if the other VM is completely idle.

We further distinguish between *preemptive* and *non-preemptive* CPU schedulers. A preemptive scheduler reruns the scheduling algorithm whenever a VM becomes runnable. If the subsequently selected VM has higher priority over the running VM, the scheduler preempts the running VM and schedules the selected VM. A non-preemptive scheduler, on the other hand, only acts on scheduling decisions when the running VM voluntarily gives up CPU. Having a preemptive scheduler is important for achieving good performance on I/O intensive workloads in shared environments. These workloads are often blocked waiting for I/O events, and their performance could suffer in presence of CPU intensive jobs if the CPU scheduler is not preemptive.

Section 2.7 presents some of the related work on memory management in virtual machine monitors. Next, we discuss the various benefits that virtualization offers. As discussed earlier, the versatility of virtualization is one of the primary drivers of its adoption.

2.2 Virtualization Benefits

Virtualization creates an additional layer of indirection between the actual physical resources and the resources visible to the virtual machines. Because this layer resides underneath the guest OS, it gives increased flexibility in how resources are managed. Furthermore, a virtual machine cleanly encapsulates and captures the state of the entire system, including the OS and all applications. This software abstraction of a physical machine is much easier to analyze and manipulate than a real physical machine, and consequently, enables several different applications that would not have been possible otherwise.

Here we outline some of the advantages of using virtual machines. This list is not exhaustive, and the advantages are not listed in any particular order.

- **Server consolidation**: As we saw earlier, server consolidation to increase utilizations and reduce infrastructure as well as power and cooling costs is one of the most attractive uses of virtualization.

- **Legacy hardware/software support**: Virtualization decouples the hardware exposed to the VMs from the actual physical hardware. VMs can therefore be used to run legacy software for which hardware does not exist or is not available [5]. In the same vein, virtual machines can be an effective interim measure on newer hardware, until operating system support for the hardware becomes robust, mature and stable.

- **Security**: The VMM can precisely control the input/output from a virtual machine. Further, if a VM crashes, it can simply be restarted. This ability to run operating systems within a secure, isolated sandbox has applications in intrusion detection and malware analysis. For instance, if a computer virus intrudes a VM, an external observer can easily determine if the VM has been infected and shut down any further network communication to/from the VM to contain the infection. The VM can then be restarted from a known "clean" state.

- **Shared hosting**: Infrastructure providers use virtual machines to create dedicated environments for clients — each client gets full control over the software stack running within the VM, and clients are unaware of other co-located clients. The strong resource isolation guarantees possible with VMs are critical for such use cases.

- **Testing and development**: VMs provide powerful system level debugging and performance monitoring. By putting debugging tools *under* the guest operating system, we can make them transparent and isolated from the guest. This allows debugging operating systems in a manner similar to regular program debugging — examining memory contents, instruction level checkpointing and replay and so on. VMs also accelerate cross-platform development: instead of building and testing applications on several different hardware platforms with different software stacks, a single physical machine hosting multiple, appropriately-configured VMs can be used to accomplish the same. Virtual machines can also be leveraged in QA testing, especially for fault-injection stress tests since multiple VMs can be used in parallel and crashed VMs can be quickly recreated from a known "golden" image.

- **Scalability and reliability**: New virtual machines can be quickly created when required. VMs can also be migrated from one physical machine to another. This flexibility in provisioning the resources allows organizations to build more agile infrastructures. For instance, recent cloud-computing offerings are using virtualization to scale applications on demand [6] — developers can use APIs to quickly instantiate new VMs as the load increases. Virtualization is also a valuable addition to the repertoire of tools and techniques for building reliable systems. A production server can be kept in sync with a suspended VM, which can transparently take over the role of the server should it fail [36]. By virtue of their design, virtual machines are ideal for localizing faults. A faulty VM may only harm itself; even if it crashes, that would not have an impact on other VMs in the system.

- **Simplified management**: Virtual machines simplify administration and provisioning of a physical infrastructure. Quickly booting pre-created VM images is much easier than a complicated setup procedure for a server. A user may even carry his/her working environment on a portable storage device for anywhere, anytime access. Keeping a set of virtual machine images updated and patched is much simpler compared to updating the same number of physical machines.

We reiterate that the above is not an exhaustive list and innovations continue to present ever new and interesting applications of virtualization. Of course, virtualization has its downsides as well. Section 9.2 discusses some of the trade-offs of using virtualization in the context of this dissertation. We now turn our attention to the virtual machine platform that we build on for all the implementation described in this dissertation.

2.3 Xen Overview

Must of the research described in this dissertations builds on top of the Xen virtual machine monitor, though the ideas remain applicable to other platforms as well. Xen [35] is an open source, Type-I hypervisor, developed initially at the University of Cambridge, UK. Figure 2.3 shows the high level architecture of Xen. When Xen boots, it creates an initial virtual machine called Domain-0. This is a privileged, control domain through which other VMs can be created and managed.

Xen pioneered an approach called *para-virtualization* where the guest OS is modified to take advantage of the knowledge that it is running on top of a VMM to optimize certain operations. For instance, the guest OS in a para-virtualized VM might directly manipulate

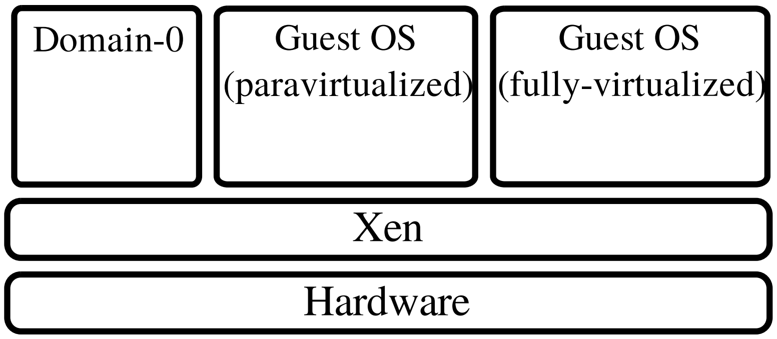


Figure 2.2: Xen is a Type-I hypervisor supporting both para-virtualized and fully-virtualized VMs.

the hardware page tables, avoiding the overhead of simulating page tables in software. Para-virtualization exports a modified (non-x86 compatible) hardware interface to the OS. This allows Xen to deliver high performance on usually expensive operations such as network I/O. However, the drawback of this approach is that the guest OS needs modifications. As a result, virtualization is no long transparent — the OS *has* to know that a virtualization layer exists — and this may not be desirable, specially when security is a concern. Further, each supported guest OS has to be modified for para-virtualization, which incurs some additional overhead. Worse yet, it might simply be impossible to adapt an OS for para-virtualization, if the source code is not available, for instance.

The primary motivation behind para-virtualization was to work around the limitations of the x86 architecture: it is well known that the x86 instruction set is not properly virtualizable [94]. However, better hardware support from both Intel [61] and AMD [26] has eliminated some of these limitations. This support allows Xen to support *fully-virtualized* VMs, that can host unmodified guest OS images. While several performance bottlenecks still remain, this approach enables running arbitrary x86-compatible OSes just with their binary images. Fully-virtualized VMs are also referred to as *hardware virtual machines* or HVMs.

In order to keep the hypervisor small and efficient, and to leverage existing code to provide broad hardware support, Xen does not contain any device drivers in the hypervisor itself. Instead, most of the hardware access, in particular the I/O devices, is relegated to Domain-0. This implies that by default, any VMs on the system — para-virtualized, or fully-virtualized — must go through Domain-0 for their I/O services. This model is called a *split I/O service model* since Domain-0 is responsible for servicing the I/O for all other VMs on the system. Of course, the mechanisms for servicing I/O requests different vastly among para-virtualized and fully-virtualized VMs. Nevertheless, this split I/O model in Xen has important ramifications for scalable multiplexing as we shall see in later chapters.

With this background on virtual machines in general and Xen, in particular, we next survey the related work in virtual machine multiplexing, network emulation, large-scale testing and memory management.

2.4 Virtual Machine Multiplexing

One of the goals in this dissertation is to address horizontal scalability by increasing the number of VMs that can be supported on a single physical machine. A notable earlier effort in this direction was the Denali isolation kernel [108], which scaled to thousands of virtual machines. However, Denali's VMs do not support general purpose operating systems — they are specialized execution environments where each VM executes a *single* application. Denali was designed specifically for supporting large scale web services. The ability to run general purpose, unmodified operating systems is critical for supporting legacy applications and to ease the transition from a physical to a virtualized infrastructure.

There is a large body of work, orthogonal to our efforts, on improving the performance overhead of virtual machine multiplexing.

2.5 Network Simulation and Emulation

The idea of changing the flow of time to explore faster networks, as used in time dilation (Chapter 4), is not a new one. Network simulators [21, 93, 96] use a similar idea; they run the network in virtual time, independent of wall-clock time. This feature allows network simulators to explore arbitrarily fast or long network pipes, but the accuracy of the experiments depends on the fidelity of the simulated code to the actual implementation. The "scalability" of the simulation are bound by compute cycles, and potentially by main memory. However, this scalability comes at the cost of realism: ideally we would like to use unmodified applications on real operating systems and network stacks. Our techniques aim to preserve realism while enabling scalable network emulation with high fidelity.

Complete machine simulators such as SimOS [90] and specialized device simulators such as DiskSim [58] have also been used for decoupling the real hardware from the hardware perceived by the operating system, in order to evaluate systems on alternate hardware configurations. However, these approaches are still fundamentally limited by the capacity of the underlying hardware. In contrast, our techniques allow pushing beyond the actual hardware capacity: time dilation combines the flexibility to explore future hardware configurations with the ability to run real-world applications on unmodified operating systems and protocol stacks.

Superficially, emulation techniques (e.g. ModelNet [102]), offer a more realistic alternative to simulation because they support running unmodified applications and operating systems. Unfortunately, such emulation is still limited by the capacity of the available physical hardware and hence is often best suited to considering wide-area network conditions (with smaller bisection bandwidths) or smaller system configurations.

There is a body of work, complementary to our effort to establish the fidelity between the dilated and real networks, that addresses the accuracy and relevance of network models. It is widely recognized that useful network models are notoriously difficult to construct [53]. Similarly, Claffy et al. [40] discussed the applicability of sampling techniques for network characterization. We would like to leverage work in this area to explore the sensitivity of time dilation to the underlying network parameters.

2.6 Testing Large Systems

One popular approach to testing complex network services is through building a simulation model of system behavior under a variety of access patterns. While such simulations are valuable, we argue that simulation is best suited for validating correctness and for understanding coarse-grained performance characteristics of certain configurations. Simulation is less well suited to detect performance problems or to capturing the effects of unexpected component interactions, failures, etc. than testing the actual system at scale on real hardware.

Further, both simulation and emulation involve trade-offs that are violate our goals of realism and scalability. Emulation is more realistic than simulation, but is still limited by the capacity of the underlying hardware. For instance, multiplexing 1,000 instances of an overlay across 50 physical machines interconnected by gigabit Ethernet may be feasible when evaluating a file sharing service on clients with cable modems. However, the same 50 machines will be incapable of emulating the network or CPU characteristics of 1,000 machines in a multi-tier network service consisting of dozens of racks and high-speed switches.

SHRiNK [87] is perhaps most closely related to DieCast in spirit. SHRiNK aims to evaluate the behavior of faster networks by simulating slower ones. For example, their “scaling hypothesis” states that the behavior of 100-Mbps flows through a 1-Gbps pipe should be similar to 10-Mbps through a 100-Mbps pipe. When this scaling hypothesis holds, it becomes possible to run simulations more quickly and with a lower memory footprint. Relative to this effort, we show how to scale fully operational computer systems, considering complex interactions among CPU, network, and disk spread across many nodes and topologies.

2.6.1 Real-World Testing

Platforms such as PlanetLab [86] and EmuLab [109] allow real-world testing of large-scale systems. PlanetLab is a federated collection of several hundred machines distributed all over the world. This infrastructure is shared by the academic research community for distributed systems and networking research. Unfortunately, because of its design and limited resources, experiments on PlanetLab are not reproducible and can not scale beyond a few hundred nodes. Further, PlanetLab only runs Linux and can not support arbitrary OS and applications.

EmuLab gives more flexibility to the users. Users can specify the desired topology and the software configuration. However, even EmuLab's scalability is constrained by the number of co-located VMs one can create on a single physical machine. Further, the network characteristics and disk performance is again limited by the capacity of the underlying hardware. DieCast is unique in its ability to support arbitrary hardware and software, as well as being able to scale beyond the capacity of the underlying hardware. It is conceivable that DieCast can be deployed on the EmuLab or PlanetLab infrastructure, integrating with some of their services for easy deployment and management.

2.6.2 Detecting Performance Anomalies

There have been a number of recent efforts to debug performance anomalies in network services, including Pinpoint [42], MagPie [34], and Project 5 [25]. Each of these initiatives analyzes the communication and computation across multiple tiers in modern Internet services to locate performance anomalies. These efforts are complementary to ours as they attempt to locate problems in deployed systems. Conversely, the goal of DieCast is to test particular software configurations at scale to locate errors before they affect a live service.

2.6.3 Modeling Internet Services

Finally, there have been many efforts to model the performance of network services to, for example, dynamically provision them in response to changing request patterns [49, 101] or to reroute requests in the face of component failures [32]. Once again, these efforts typically target already running services relative to our goal of testing service configurations. Alternatively, such modeling could be used to feed simulations of system behavior or to verify at a coarse granularity DieCast performance predictions.

2.7 Memory Management

Difference Engine builds upon a large body of previous work in page sharing, memory compression, and delta encoding. In each case, we attempt to leverage existing approaches where appropriate.

2.7.1 Page Sharing

Two common approaches in the literature for finding redundant pages are content-based page sharing (exemplified by VMware ESX server [110]) and explicitly tracking page changes to build knowledge of identical pages (e.g., the "transparent page sharing" used in Disco [27]). Transparent page sharing can be more efficient, but requires several hooks and modifications into the guest OS, not required by ESX server or by Difference Engine. This property is desirable for supporting unmodified OS images.

To find sharing candidates, ESX hashes contents of each page and uses hash collisions to identify potential duplicates. Both ESX server and Difference Engine perform a byte-by-byte comparison once a hash match is found before actually sharing the page.

Once shared, pages in our system can manage updates in a copy-on-write fashion, as in Disco and ESX server. We build upon earlier work on *flash cloning* [103] of VMs, which allows new VMs to be cloned from an existing VM in milliseconds; as the newly created VM writes to its memory, it is given private copies of the shared pages. An extension by Kloster et al. studied full-page sharing in Xen [68] and we build upon this experience, adding support for fully virtualized (HVM) guests, integration with the global clock, and numerous bug fixes and optimizations.

2.7.2 Delta Encoding

Our initial investigations into page similarity were inspired by research in leveraging similarity across files in large file systems. There are two issues we needed to address: how to find a suitable candidate page to construct a patch against and, once a candidate page is found, how to construct and store the patch.

In the GLIMPSE system [79], Manber proposed computing Rabin fingerprints over fixed-size blocks at multiple offsets in a file. Similar files will then share some fingerprints. One way of finding a candidate page is to pick the one with the maximum number of common fingerprints. However, in a dynamically evolving virtual memory system, this approach does not scale well since every time a page changes its fingerprints must be recomputed as well. Further, it is inefficient to find the maximal intersecting set from among a large number of candidate pages.

Broder adapted Manber's approach to the problem of identifying documents (in this case, Web pages) that are nearly identical using a combination of Rabin fingerprints and sampling based on minimum values under a set of random permutations [30]. His paper also contains a general discussion of how thresholds should be set for inferring document similarity based on the number of fingerprints (or sets of fingerprints) in common.

While these techniques can be used to identify similar files, they do not address how to efficiently encode the differences. Douglis and Iyengar explored using Rabin fingerprints and delta encoding to compress similar files in the DERD system [45], but only considered whole files. Kulkarni *et al.* [66] extended the DERD scheme to exploit similarity at the block level. In this spirit, Difference Engine also tries to exploit memory redundancy at several different granularities.

2.7.3 Memory Compression

Compressing memory is not a new idea. Douglis *et al.* [44] implemented memory compression in the Sprite operating system with mixed results. In their experience, memory compression was sometimes beneficial, but at other times the performance overhead outweighed the memory savings. Subsequently, Wilson *et al.* argued Douglis' mixed results were primarily due to slow hardware [111]. They also developed new compression algorithms that exploited the inherent structure present in virtual memory, whereas earlier systems used general purpose compression algorithms.

Despite its mixed history, several operating systems have dabbled with in-memory compression. A company called Ram Doubler launched a product that promised to "double the RAM" on Apple Macintoshes in the early '90s [72]. Tuduce *et al.* [100] implemented a compressed cache for Linux that adaptively manages the amount of physical memory devoted to compressed pages using a simple algorithm shown to be effective across a wide variety of workloads. Castro *et al.* present an alternative adaptive scheme where the compressed area is resized based on estimating whether the page would be on disk if compression were not used [50].

In the next chapter, we introduce DieCast: our framework for testing large systems accurately, using a much smaller infrastructure. In the process, we outline the requirements of such a framework and the building blocks that constitute the system.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2006. Gupta, Diwaker; Yocum, Kenneth; McNett, Marvin; Snoeren, Alex C.; Vahdat, Amin; Voelker, Geoffrey M. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2008. Gupta, Diwaker; Vishwanath, Kashi V.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI) 2008. Gupta, Diwaker; Lee, Sangmin; Vrable, Michael; Savage, Stefan; Snoeren, Alex C.; Varghese, George; Voelker, Geoffrey M.; Vahdat, Amin. The dissertation author is the primary investigator and author of this paper.

Chapter 3

A Framework for Large Scale Testing

We use the problem of large-scale system testing (as described in Chapter 1) to present our approach for addressing the challenges of vertical and horizontal scalability. We wish to develop a testing methodology and architecture that can accurately predict the behavior of modern network services while employing an order of magnitude fewer hardware resources. This chapter describes the architecture for DieCast— a framework for accurate, scalable testing of large systems. Such a framework should capture the end-to-end behavior of the target system, including the application level performance metrics as well as the low-level system behavior. As we discussed in the previous chapter, while emulation and simulation are valuable techniques for evaluating services in their own right, they do not provide the level of realism and scalability that we desire.

We start with a general overview of our approach to scaling a system down to a target test harness. In particular we discuss the desired goals of such a system and why a virtual machine based infrastructure makes sense in this context. Next we discuss the building blocks of the architecture and conclude with the key missing building blocks that will be the subject matter of subsequent chapters.

3.1 Overview

For DieCast to be useful and reliable, it should meet the following goals:

- **Fidelity**: DieCast must accurately mimic the behavior of the target system. Critically, besides replicating the steady-state behavior of the system, DieCast should also capture any faulty behavior or performance anomalies. Not only do we want to match the application performance, we also want to capture and preserve the system level behavior such as resource utilization profiles.

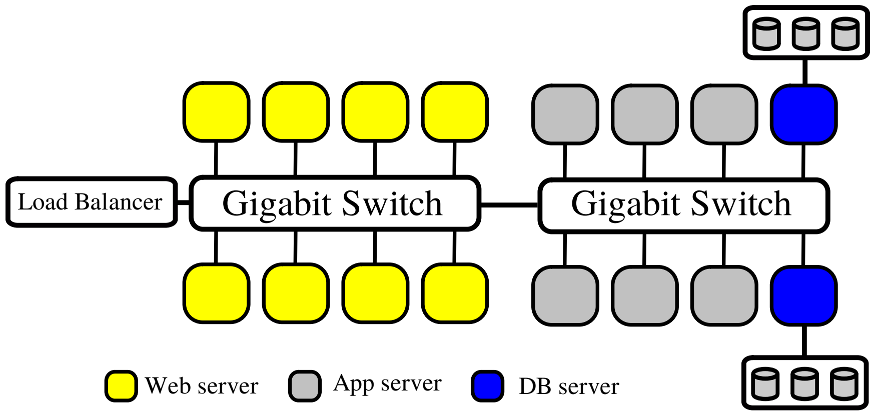
- **Efficiency**: Efficiency is an overloaded term in the literature. We define efficiency as the ratio of the size of the original system to the size of the test harness. Thus, a system with high efficiency will be able to replicate and test the original system using fewer resources than a low efficiency system.

- **Reproducibility**: The ability to conduct repeatable experiments in a controlled environment is crucial for isolating faults and performance problems. Thus, live deployments on platforms such as PlanetLab [86] are not suitable since we have no control over the background traffic, for instance.

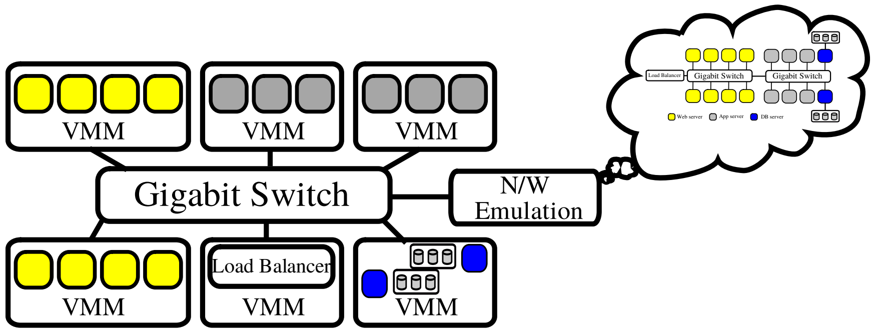
The ideal test environment would simply duplicate the entire infrastructure, including all the physical machines, networking hardware and software configuration of the original system. However, this approach has low efficiency as defined above. We want to reproduce similar conditions, except using far fewer machines. One natural approach to reducing the number of machines is to simply encapsulate each physical machine in the original system into a VM, and then multiplex these VMs on a fewer physical machines. But this naïve server consolidation does not work because each individual VM might have a very different resource capacity than its corresponding physical machine in the original system, and hence will behave differently, violating fidelity.

Figure 3.1 gives an overview of our approach. On the top (Figure 3.1a) is an abstract depiction of a network service. A load balancing switch sits in front of the service and redirects requests among a set of front-end HTTP servers. These requests may in turn travel to a middle tier of application servers, who may query a storage tier consisting of databases or network attached storage returning a final result to the client who made the original request.

Figure 3.1b shows how a target service might be scaled with DieCast. Clearly, for efficiency we need to multiplex several components of the original system onto fewer machines. This multiplexing might be done in one of many ways — for instance, a single machine may host both the web server and the database server processes. However, virtual machines a much cleaner abstraction for this kind of multiplexing. Not only do VMs offer strong isolation, accurate resource allocation but they also make it fairly straight forward to go from the original system to the test harness. Thus, we encapsulate all nodes from the original service in virtual machines and multiplex several of these VMs onto physical machines in the test harness.



(a) Original System



(b) Test System

Figure 3.1: Scaling a network service to the DieCast infrastructure.

However, this multiplexing destroys the original network topology. DieCast leverages network emulation to recreate the network topology and characteristics of the original system. But for a faithful reproduction of the original system, apart from the network topology, we require *resource equivalence* for each physical machine of the original system: each VM in the test harness should perceive the exact same resources and exhibit the precise behavior of the corresponding physical machine in the original system.

As we saw in Chapter 1, multiplexing causes resource partitioning, which violates resource equivalence. Consider a simple distributed system comprising five, identical physical machines. Each machine has a 3-GHz processor, 1 Gbps of network bandwidth and 15 Mbps of read throughput on the disk. In a DieCast-scaled system, we create five virtual machines corresponding to each physical machine in the original system. For efficiency, let us say we multiplex these virtual machines on a single physical machine. If this physical machine has the same specifications as the machines in the original system, and all five share the resources equally, then each VM will effectively get the equivalent of a 600-MHz processor, 400-MB RAM, 200 Mbps bandwidth on the network and 3 Mbps disk throughput. Clearly then, each of these VMs will behave and perform very differently than their physical counterparts. For DieCast to accurately replicate system behavior and capacity, we ideally want each of these virtual machines to perceive the same amount of resources that a physical machine in the original system did.

The actual physical resources in a given test harness are finite, and thus limited. We instead attack the problem of resource partitioning by increasing the *perceived* resource capacity of VMs. For the remainder of this chapter, assume that we have at our disposal a mechanism to uniformly scale various resources in the system such as CPU, network bandwidth and disk I/O throughput, by a configurable scaling factor. We call this mechanism *time dilation* and the scaling factor the *time dilation factor* (TDF). Thus in the above example, if we were to multiplex the virtual machines with a TDF of 5, then each virtual machine would *perceive* a 3-GHz CPU, 1-Gbps network bandwidth and 15-Mbps disk throughput for reads. In other words, it will perceive the same resource capacity as a physical machine in the original system. The next chapter describes time dilation and related issues in further detail.

The overall goal is to improve predictive power. That is, runs with DieCast on smaller machine configurations should accurately predict the performance and fault tolerance characteristics of some larger production system. In this manner, system developers may experiment with changes to system architecture, network topology, software upgrades, and new functionality before deploying them in production. Successful runs with DieCast should improve confidence that any changes to the target service will be successfully deployed. Below, we discuss the steps in applying our general approach to applying DieCast scaling to target systems.

3.2 Choosing the Scaling Factor

The first question to address is the desired scaling factor. One use of DieCast is to reproduce the scale of an original service in a test cluster. Another application is to scale existing test harnesses to achieve more realism than possible from the raw hardware. For instance, if 100 nodes are already available for testing, then DieCast might be employed to scale to a thousand-node system with a more complex communication topology. While the DieCast system may still fall short of the scale of the original service, it can provide more meaningful approximations under more intense workloads and failure conditions than might have otherwise been possible.

Overall, the goal is to pick the largest scaling factor possible while still obtaining accurate predictions from DieCast, since the prediction accuracy will naturally degrade with increasing scaling factors. This maximum scaling factor depends on the characteristics of the target system. Section 6.6 highlights the potential limitations of DieCast scaling. In general, scaling accuracy will degrade with:

1. application sensitivity to the fine-grained timing behavior of external hardware devices;
2. capacity-constrained physical resources; and
3. system devices not amenable to virtualization.

In the first category, application interaction with I/O devices may depend on the exact timing of requests and responses. Consider for instance a fine-grained parallel application that assumes all remote instances are co-scheduled. A DieCast run may mis-predict performance if target nodes are not scheduled at the time of a message transmission to respond to a blocking read operation. If we could interleave at the granularity of individual instructions, then this would not be an issue. However, context switching among virtual machines means that we must pick time slices on the order of milliseconds. Second, DieCast cannot scale the capacity of hardware components such as main memory, processor caches, and disk. Finally, the original service may contain devices such as load balancing switches that are not amenable to virtualization or dilation. Even with these caveats, we have successfully applied scaling factors of 10 to a variety of services with near-perfect accuracy as discussed in Chapter 6.

Of the above limitations to scaling, we consider capacity limits for main memory and disk to be most significant. However, we do not believe this to be a fundamental limitation. In particular, secondary storage is readily available and the price/byte continues to decrease. Thus, one partial solution is to configure the test system with more memory and storage than the original system. While this approach will reduce some of the economic benefits of our approach, it will not erase them. For instance, doubling a machine's memory will not typically double its hardware cost. More importantly, it will not substantially increase the typically dominant human cost of administering a given test infrastructure because the number of required administrators for a given test harness usually grows with the number of machines in the system rather than with the total memory of the system.

With respect to main memory, ongoing research in VMM architectures have the potential to reclaim some of the memory [110] and storage overhead [107] associated with multiplexing VMs on a single physical machine. For instance, four nearly identically configured Linux machines running the same web server will overlap significantly in terms of their memory and storage footprints. Similarly, consider an Internet service that replicates content for improved capacity and availability. When scaling the service down, multiple machines from the original configuration may be assigned to a single physical machine. A VMM capable of detecting and exploiting available redundancy could significantly reduce the incremental storage overhead of multiplexing multiple VMs. In effect, this increases the horizontal scalability on a system. We have built a system (Chapter 7) that saves a factor of two to three more memory than the current state of the art.

Initially, we view the task of choosing the scaling factor to be both service-specific and requiring some validation. However, it should be feasible to partially automate this process. We leave this to future work.

3.3 Cataloging the Original System

The next task is to configure the appropriate virtual machine images onto our test infrastructure. Maintaining a catalog of the hardware and software configuration that comprises an Internet service is challenging in its own right. However, for the purposes of this dissertation, we assume that such a catalog is available. This catalog would consist of all of the hardware making up the service, the network topology, and the software configuration of each node.

The software configuration includes the operating system, installed packages and applications, and the initialization sequence run on each node after booting.

The original service software may or may not run on top of virtual machines. However, given the increasing benefits of employing virtual machines in data centers for service configuration and management and the popularity of VM-based appliances that are pre-configured to run particular services [23], we assume that the original service is in fact VM-based. This assumption is not critical to our approach but it also partially addresses any baseline performance differential between a node running on bare hardware in the original service and the same node running on a virtual machine in the test system. Tools such as VMware's P2V Assistant [106] can automate the process of converting an existing physical machine to a VM image.

We further require a tool that ideally automates the process of determining the characteristics of a given networked service. Important characteristics include:

- the physical machines that makes up the service (processor, memory, disk capacity, etc.),
- the software configuration of each machine (operating system version, application software running on each machine, etc.),
- the network interconnectivity of the target service (switches, routers, routes through the installation, etc.), and
- any special hardware in the service such as load balancing switches, firewalls, and intrusion detection systems.

Of course, accurately characterizing and inventorying a large-scale network service in this manner is a difficult challenge in and of itself. Ideally, an automated tool would run periodically to perform *introspection* of the network services, building an accurate model of service characteristics and interconnectivity. Many networked services employ a combination of manual and automated techniques to build a database containing this information. The quality of the models that DieCast builds will improve with our ability to extract network service characteristics.

3.4 Workload Generation

Once DieCast has prepared the test system to be resource equivalent to the original system, we can subject it to an appropriate workload. These workloads will in general be application-specific. For instance, Monkey [38] shows how to replay a measured TCP request stream sent to a large-scale network service. For this dissertation, we use application-specific workload generators where available and in other cases write our own workload generators that both capture normal behavior as well as stress the service under extreme conditions.

To maintain a target scaling factor, clients should also ideally run in DieCast-scaled virtual machines. This approach has the added benefit of allowing us to subject a test service to a high level of perceived-load using relatively few resources. Thus, DieCast scales not only the capacity of the test harness but also the workload generation infrastructure.

3.5 Network Emulation

DieCast configures VMs to communicate through a network emulator to reproduce the characteristics of the original system topology. Using scripts similar to those employed in the original system we can then initialize the test system and subject it to appropriate workloads and fault-loads to evaluate system behavior. The next step in the configuration process is to match the network configuration of the original service using network emulation. We configure all VMs in the test system to route all their communication through our emulation environment. Note that DieCast is not tied to any particular emulation technology: we have successfully used DieCast with Dummynet [89], ModelNet [102] and Netem [15] where appropriate.

It is likely that the bisection bandwidth of the original service topology will be larger than that available in the test system. Thus somehow we need to to support higher bisection bandwidths than the capacity of the underlying physical hardware.

3.6 Configuring the Virtual Machines

With an understanding of appropriate scaling factors and a catalog of the original service configuration, DieCast then configures individual physical machines in the test system with multiple VM images reflecting, ideally, a one-to-one map between physical machines in the original system and virtual machines in the test system. With a scaling factor of 10, each physical node in the target system would host 10 virtual machines. The mapping from physical machines to virtual machines should account for: similarity in software configurations, per-VM memory and disk requirements and the capacity of the hardware in the original and test system. In general, a solver may be employed to determine a near-optimal matching [92]. However, given the VM migration capabilities of modern VMMs and DieCast's controlled network emulation environment, the actual location of a VM is not as significant as in the original system. For the relatively modest services that we have considered to date, we perform this mapping by hand.

We alluded earlier to our technique of time dilation to address resource partitioning and it is the subject matter for the next chapter. However, time dilation scales all the resources *uniformly*, that is, by the same scaling factor. Such uniform scaling works fine in cases where all system components are homogeneous, but to deal with heterogeneous configurations, we require some additional mechanisms. Let us look at examples of each scenario.

Consider a physical machine hosting 10 VMs. We would like to configure the VMS such that each VM appears to have the processing power of the entire original machine available to it. To address resource partitioning, say we run the VMs with a scaling factor of 10. With this simple strategy, however, if nine of the virtual machines were mostly idle at any point in time, the tenth virtual machine would appear to have 10 times the processing power of the original machine (all the power of the original machine, dilated by a factor of 10). To address this, DieCast would run each VM with a scaling factor of 10, but allocate each VM only 10% of the actual physical resource. We only explicitly scale CPU and disk I/O latency on the host; scaling of network I/O happens via network emulation as described earlier.

However, for heterogeneous systems, more flexible mechanisms are needed. For instance, imagine that in the original system each physical machine is equipped with 2-GHz processors but the test harness is made up of older, 1-GHz machines. However, both the original system and test harness have identical disk drives. Now, to achieve a scaling factor of two, we need to multiplex two VMs on each test machine. However, with a TDF of two, each dilated VM will still only perceive a 1-GHz processor where as we would like the VMs to perceive 2-GHz CPUs. This can be done by using a TDF of four, but in this case the disk drives appear twice as fast under DieCast. What we really need are mechanisms for independently scaling CPU, disk and network under a dilated time frame. In the example above, we could scale down the disk by a factor of two in the dilated time frame such that the perceived disk characteristics are preserved. These mechanisms are the subject of Chapter 5.

Using these knobs and an appropriate scaling factor, DieCast configures the VMs such that each VM appears to have resources identical to a physical machine in the original system. In the homogeneous case, we assume that the processing power (and other resources) of the 10 physical machines running in the original system is identical to the processing power of the one machine in the test harness. This assumption makes it straightforward to determine the amount of resources to allocate to each virtual machine, simply based on the dilation factor. With homogeneous machines and a scaling factor of sf, DieCast configures the VMs to allocate $(1/sf)$ resources to each virtual machine independent of the number of virtual machines hosted on a particular physical machine. In general however, there will be significant heterogeneity in both the original and test service machines. Thus, we must account for the resource differential when assigning resources to individual components. For instance if a machine in the original configuration is r times more powerful (or less powerful for $r < 1$) than the corresponding test machine, the scheduler would assign (r/sf) resources to the corresponding virtual machine.

Assigning a single number to capture the performance differential between two machines will introduce some error in test results, for instance because processor performance, cache size, I/O bus speed, etc. may all scale independently. However, we believe that for small performance differentials, this error is acceptable.

3.7 Summary

In this chapter we have provided an overview of the DieCast approach. Our goal is to create a hi-fidelity replica of the original system using fewer resources in a controlled environment. For efficiency, we encapsulate each physical machine in the original system in its own virtual machine, and then we consolidate these VMs onto fewer physical machines. Before that we need a detailed specification of the hardware and software configuration of the system. Several tools exist to make it easy to convert physical machines to VM images.

However, multiplexing causes resource partitioning, so a VM in the DieCast-scaled system might have a resource capacity very different than its physical counterpart in the original system. For preserving fidelity, we want to maintain the invariant of resource equivalence: each VM in the DieCast-scaled system should perceive resources similar to the corresponding physical machine in the original system.

In the next chapter we describe time dilation, a technique that allows increasing the perceived resource capacity of individual VMs by a configurable scaling factor. However, this alone is not enough to preserve resource equivalence. Machines in the test harness might have very different hardware configurations than machines in the original system. Further, machines in both the original system and the test harness might not be homogeneous. Consequently, we must be able to precisely configure the resources available to each VM, such that after time dilation, the perceived resource capacity matches with the original system. Chapter 5 describes the mechanisms to independently control allocation of various system resources to individual VMs.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2008. Gupta, Diwaker; Vishwanath, Kashi V.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 4

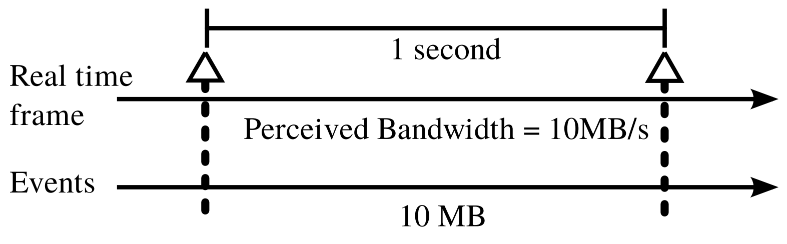
Time Dilation

A critical building block of DieCast is the ability to scale up the perceived resource capacity of a virtual machine. This chapter describes the concept and design of time dilation. We conclude the chapter with the details of our implementation of time dilation in Xen and a discussion of some of the limitations of this technique.

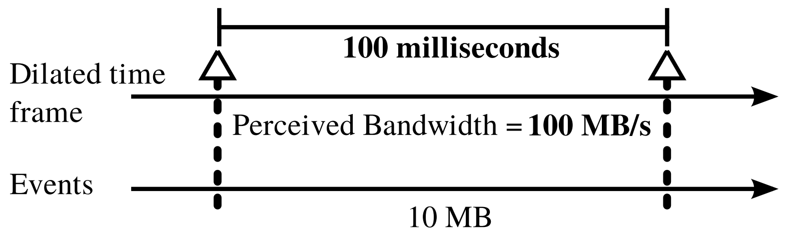
4.1 Concept

The key insight behind time dilation is that we can trade off time for other resources in the system. By providing the illusion to an operating system and its applications that time is passing at a rate different from real time, we can manipulate the perceived capacity of the system. For example, we may wish to convince a system that, for every ten seconds of wall-clock time, only one second of time passes in the operating system's dilated time frame. Time dilation does not, however, change the arrival rate of physical events such as those from I/O devices like a network interface or disk controller. Hence, in this example, from the operating system's perspective, physical resources appear 10 times faster: in particular, data arriving from a network interface at a physical rate of 10 MBps appears to the OS to be arriving at 100 MBps.

We refer to the ratio between the rate at which time passes in the physical world to the operating system's perception of time as the time dilation factor, or TDF; a TDF greater than one indicates the external world appears faster than it really is. Figure 4.1 illustrates the difference between an operating system in the real time frame (Figure 4.1a) and an OS with a TDF of 10 (Figure 4.1b). The same period of physical time passes for both operating systems.



(a) Real time frame



(b) Dilated time frame (TDF 10)

Figure 4.1: Compare a system operating in real time (top) with a system running with a TDF of 10 (bottom). Note that time dilation does not affect the rate of external events, such as network packet arrival.

Each OS receives the same amount of data over the network (10 MB), the only difference is the time interval over which this data is received as *perceived* by the OS and the applications. The undilated OS perceives a bandwidth of 10 MBps, while the OS in the dilated time frame perceives a much faster bandwidth of 100 MBps.

This observation is critical: physical devices such as the network continue to deliver events at the same rate to both OSes. The dilated OS, therefore, perceives ten times the network I/O rate because it experiences only one tenth the delay between I/O events. Similarly, time dilation scales the perceived available processing power as well. A system will experience TDF times as many cycles per perceived second from the processor. Such CPU scaling is particularly relevant to CPU-bound network processing because the number of cycles available to each arriving byte remains constant. For instance, a machine with TDF of ten sees a ten times faster network, but would also experience a tenfold increase in CPU cycles per second.

Generalizing, we observe that time dilation impacts all *temporal* or *streaming* resources. That is, resources that have a time based rate associated with them, such as CPU cycles per second or network bandwidth or disk bandwidth. On the other hand, *spatial* resources — those that can be statically partitioned and do not have a time based rate associated with them, such as main memory or disk capacity — are not affected by time dilation.

Thus time dilation effectively scales up the perceived resource capacity of any given physical machine. As mentioned in the first chapter, this scaling up of resources enables several interesting applications, including extrapolation to future scenarios and scalable network emulation. We will see how this helps DieCast in a subsequent chapter.

4.2 Implementation

Time dilation must encapsulate the OS in an arbitrary time frame, distinct from the real time frame. Note that an operating system relies on various time sources to keep track of time. Most applications running within the OS rely on the operating system's timekeeping facilities for keeping track of time in user space. Hence, manipulating the operating system's perception of time is necessary and in most cases, sufficient, to implement time dilation. In order to do this, we need to interpose between each time source that the OS consumes, and manipulate it appropriately before it is used by the OS. If all time sources are appropriately adjusted, the OS will automatically perceive a different time frame.

For instance, suppose a regular, undilated OS receives 100 timer interrupts every second (that is, it runs with a 100-Hz clock). Then, the same OS running with a TDF of 10 should only receive 10 timer interrupts every second of *real* time, such that the perceived rate of timer interrupts is still 100 per second of *dilated* time. Other common time sources are: special hardware registers such as the Time Stamp Counter (TSC) on x86 platforms, on board crystals and platform timers such as the High Performance Event Timer (HPET) or the Programmable Internet Timer (PIT), information stored in the BIOS, external time sources (such as NTP) and so on.

In terms of actually building an implementation based on the above strategy, one approach would be to implement time dilation directly in the operating system. In other words, the OS would take the time dilation factor as input (perhaps as a boot time parameter) and internally manipulate the various time sources. As an example, the OS might choose to ignore some of the timer interrupts it receives from the underlying hardware. This approach is problematic for several reasons:

- It violates transparency, because the OS is clearly aware of time dilation.
- Each supported OS would have to be modified individually.

- Some modifications — such has manipulating the frequency of timer interrupts — are much harder, if not impossible, to implement in hardware.

This dissertations pursues the alternative approach of implementing time dilation in the virtual machine monitor. VMMs already provide a clean abstraction of the underlying hardware to the guest OS, including several standard time sources that most OSes use. By adjust the time sources at these interfaces, time dilation immediately becomes available to all OSes supported by the VMM. This saves us from reinventing the wheel for each different OS, and also allows from a completely transparent implementation. Additionally, modifications are contained entirely in software, since the VMM has software based mechanisms for functionality that might otherwise usually reside in hardware, such as timers and interrupt delivery.

Of course, a virtualization-based implementation might not always be possible. For instance, the OS might not be virtualizable, or the VMM might not run on a particular hardware platform. In such cases, nothing precludes a direct implementation of time dilation, and in fact we *have* ported time dilation to non-virtualized platforms (see Section 6.5). Alternative implementation targets for time dilation include directly modifying the operating system, simulation packages, and emulation environments.

Critical to time dilation is a VMM's ability to modify the perception of time within a guest OS. Fortunately, VMMs must already perform this functionality, for example, because a guest OS may develop a backlog of "lost ticks" if it is not scheduled on the physical processor when it is due to receive a timer interrupt. VMMs typically periodically synchronize the guest OS time with the physical machine's clock. One challenge is that operating systems often use multiple time sources for better precision. For example, Linux uses up to five different time sources [71]. Exposing so many different mechanisms for time keeping in virtual machines becomes challenging (see [105] for a discussion).

To address these requirements, we implemented a time dilation prototype using the Xen VMM. Though our implementation is Xen-specific, we believe the concepts can apply to other virtual machine environments. Our current implementation supports Xen versions 2.0.7, 3.0.4 and 3.1 (both para-virtualized and fully-virtualized VMs).

As discussed below, our modifications to Xen are compact and portable, giving us confidence that our techniques will be applicable to any operating system that Xen supports. In some sense, time dilation is free in many simulation packages: extrapolating to future scenarios might be as simple as setting appropriate values on particular parameters such as bandwidth on network links. However, we explicitly target running unmodified applications and operating systems for necessary realism. Finally, while network emulation does allow experimentation with a range of network conditions, it is necessarily limited by the performance of currently available hardware. For this reason, time dilation is a valuable complement to network emulation, allowing an experimenter to easily extrapolate evaluations to future, faster environments.

We now give a brief overview of time keeping in Xen, describe our modifications to it, and discus the applicability of time dilations to other virtualization platforms. Our modifications to Xen are small: in all, we added/modified approximately 1,100 lines of C and Python code. More than 50% of our changes are to non-critical tools and utilities; the core changes to Xen and XenoLinux are less than 500 lines of code. Our modifications are less than 0.5% of the base code size of each component.

4.2.1 Time Flow in Xen

The Xen VMM exposes two notions of time to VMs. *Real time* is the number of nanoseconds since boot. *Wall-clock time* is the traditional Unix time-since-epoch (midnight, January 1, 1970 UTC). Xen also delivers periodic timer interrupts to the VM to support the time keeping mechanisms within the guest OS.

While Xen allows the guest OS to maintain and update its own notion of time via an external time source (such as NTP), the guest OS often relies solely on Xen to maintain accurate time. Real and wall-clock time pass between the Xen hypervisor and the guest operating system via a shared data structure. There is one data structure per VM written by the VMM and read by the guest OS.

The guest operating system updates its local time values on demand using the shared data structure — for instance, when servicing timer interrupts or calling `getttimeofday`. However, the VMM updates the shared data structure only at certain discrete events, and thus it may not always contain the current value. In particular, the VMM updates the shared data structure when it delivers a timer interrupt to the VM or schedules the VM to run on an available CPU.

As described in Chapter 2, Xen uses *para-virtualization* to achieve scalable performance with virtual machines without sacrificing functionality or isolation. With para-virtualization, Xen does not provide a perfect virtualization layer. Instead, it exposes some features of the underlying physical hardware to gain significant performance benefits. For instance, on the x86 architecture, Xen allows guest OSes (for our tests, we use Linux as our

Table 4.1: Basic dilation summary.

Variable	**Original**	**Dilated**
Real time	`stime_irq`	`stime_irq/tdf`
Wall-clock	`wc_sec, wc_usec`	`wc_sec/tdf, wc_usec/tdf`
Timer interrupts	`HZ/sec`	`(HZ/tdf)/sec`

guest OS) to read the Time-Stamp Count (TSC) register directly from hardware (via the `RDTSC` instruction). The TSC register stores the number of clock cycles since boot and is incremented on every CPU cycle. The guest OS reads the TSC to maintain accurate time between timer interrupts. By contrast, kernel variables such as Linux `jiffies` or BSD `ticks` only advance on timer interrupts.

Newer versions of Xen, however, leverage hardware support for virtualization allowing Xen to boot unmodified operating systems — this is required to boot proprietary operating systems such as Windows. Recent processors by both Intel and AMD provide this support, and in fact even allow the TSC to be virtualized in hardware. Our implementation appropriately scales the TSC for both para-virtualized and fully-virtualized domains. For para-virtualized VMs, the value of the TSC is scaled *within* the guest OS (thus guest OS modifications are needed). For HVM VMs, the hypervisor itself exposes the scaled value to the VM, hence no guest modifications are needed.

4.2.2 Modifications to the Xen hypervisor

Our modified VMM maintains a TDF variable for each hosted VM. For our applications, we are concerned with the relative passage of time rather than the absolute value of real time; in particular, we allow—indeed, require—that the host's view of wall-clock time diverge from reality. Thus the TDF divides both real and wall-clock time.

We modify two aspects of the Xen hypervisor. First we extend the shared data structure to include the TDF field. Our modified Xen tools, such as `xm`, allow specifying a positive, integral value for the TDF on VM creation. When the hypervisor updates the shared data structure, it uses this TDF value to modify real and wall clock time. In this way, real time is never exposed to the guest OS through the shared data structure. This design also allows us to provide each VM with an independent time frame.

Dilation also impacts the frequency of timer interrupts delivered to the VM. The VMM controls the frequency of timer interrupts delivered to an undilated VM (timer interrupts/second); in most OS's a HZ variable, set at compile time, defines the number of timer interrupts delivered per second. For transparency, we need to maintain the invariant that HZ accurately reflects the number of timer interrupts delivered to the VM during a second of dilated time. Without adjusting timer interrupt frequency, the VMM will deliver TDF-times too many interrupts. For example, the VMM will deliver HZ interrupts in one physical time second, which will look to the dilated VMM as $HZ/(second/TDF) = TDF \times HZ$. Instead, we reduce the number of interrupts a VM receives by a factor of TDF. Table 4.1 summarizes the discussion so far.

Finally, Xen runs with a default HZ value of 100, and configures guests with the same value. However, $HZ = 100$ gives only a 10-ms precision on system timer events. In contrast, current 2.6 series of Linux kernels uses a HZ value of 1000 by default—the CPU overhead is not significant, but the accuracy gains are tenfold. This increase in accuracy is desirable for time dilation because it enables guests to measure time accurately even in the dilated time frame. Thus, we increase the HZ value to 1000 in both Xen and the guest OS.

The changes described thus far are required regardless of the type of virtual machine — para-virtualized or fully-virtualized. These modifications implement the core functionality required in the hypervisor to support time dilation for either type of VM. Next we describe the modifications required to a para-virtualized guest OS kernel followed by our hypervisor extensions specific to the HVM support.

4.2.3 Modifications to XenoLinux

One goal of our implementation was to minimize required modifications to the guest OS. Because the VMM appropriately updates the shared data structure, one primary aspect of OS time-keeping is already addressed. We further modify the guest OS to read an appropriately scaled version of the hardware Time Stamp Counter (TSC). XenoLinux now reads the TDF from the shared data structure and adjusts the TSC value in the function `get_offset_tsc`.

The Xen hypervisor also provides guest OS programmable alarm timers. Our last modification to the guest OS adjusts the programmable timer events. Because guests specify timeout values in their dilated time frames, we must scale the timeout value back to physical time. Otherwise they may set the timer for the wrong (possibly past) time.

The modifications described here are only relevant for para-virtualized VM images which are running a modified kernel. While we only describe our modifications to the Xeno-Linux kernel here, we have explored modifying other guest OSes as well and the modifications are generally quite small.

Note that the para-virtualized virtual machines already require significant OS modifications, and our additional modifications to support time dilation are negligible in comparison. Next, we describe time dilation support for fully virtualized VMs where all our modifications are contained within the hypervisor and unmodified operating systems can be run as is.

4.2.4 Support for OS Diversity

The para-virtualized version of time dilation is quite limited in its utility because of the following reasons: it only supports Linux as the guest OS, and the guest kernel requires modifications to fully support pervasive time dilation. For instance, without the modifications to read an appropriately scaled TSC, an application might be able to peek "outside the box" of the dilated time frame to infer the real time frame, thus violating transparency. Generalizing to other platforms would have required code modifications to the respective OS. To be widely applicable, DieCast (and hence time dilation) must support a variety of operating systems.

To address these limitations, we ported time dilation to support fully-virtualized VMs, enabling support for unmodified OS images. While Xen support for fully virtualized VMs differs significantly from the para-virtualized VM support in several key areas such as I/O emulation, access to hardware registers, and time management, the general strategy for implementation remains the same: we want to intercept all sources of time and scale them. In particular, our implementation scales the Programmable Interrupt Timer (PIT), the TSC register on x86 platforms, the Real Time Clock (RTC), the ACPI power management timer and the High Performance Event Timer (HPET).

4.2.5 Time Dilation on Other Platforms

Architectures: Our implementation should work on all platforms supported by Xen. One remark regarding transparency of time dilation to user applications in para-virtualized VMs on the x86 platform is in order: recall that we intercept calls to read the TSC within the guest kernel. However, since the `RDTSC` instruction is not a privileged x86 instruction, guest user applications might still issue assembly instructions to read the hardware TSC register. It is possible to work around this by watching the instruction stream emanating from a VM and trapping to the VMM on a `RDTSC` instruction, and then returning the appropriate value to the VM. However, this approach would go against Xen's para-virtualization philosophy and would results in an unacceptable performance slowdown. An alternative approach would be to do binary rewriting (as done by VMware ESX server).

This is a non-issue for fully virtualized VMs on platforms with appropriate hardware support. Both Intel and AMD processors have hardware support for trapping the `RDTSC` instruction making it easy to virtualize within the hypervisor itself, making the *RDTSC* interposition completely transparent to the guest OS and applications within, even when using assembly instructions.

VMMs: The only fundamental requirement from a VMM for supporting time dilation is that it have mechanisms to update/modify time perceived by a VM. As mentioned earlier, due to the difficulties in maintaining time within a VM, all VMMs already have similar mechanisms so that they can periodically bring the guest OS time in sync with real time. For instance, VMware has explicit support for keeping VMs in a "fictitious time frame" that is at a constant offset from real time [105]. Thus, it should be straightforward to implement time dilation for other VMMs. In fact, for hosted VMMs (Type II VMMs) such as Qemu and Bochs, it is even easier to implement time dilation since they usually operate completely in the user-space.

Operating systems: Our para-virtualized implementation provides dilation support only for Linux. Our experience so far and a preliminary inspection of the code for other guest OSes indicate that all of the guest OSes that Xen supports can be easily modified to support time dilation. In fact, time dilation can also be implemented on operating systems that are not supported by Xen. Section 6.5 describes our modifications to a non-virtualized proprietary operating system to support time dilation. See section 9.2 for a discussion on the question of whether the VMM or the OS is the right place for implementing time dilation.

4.3 Limitations

This section discusses some of the limitations of time dilation. One obvious limitation is time itself: since time dilation slows down the passage of time within the OS, an hour long experiment (in the OS's time frame) would run for ten hours for a TDF of 10. Real-life experiments running for hours are not uncommon, so the time required to run experiments at high dilation factors is substantial. Below we discuss other, more subtle limitations.

4.3.1 Pervasiveness and Fidelity

Time dilation uniformly scales the perceived performance of all system devices, including network bandwidth, perceived CPU processing power, and disk and memory I/O.

Unfortunately, scaling all aspects of the physical world is unlikely to be useful: a researcher may wish to focus on scaling certain features (e.g., network bandwidth) while leaving others constant. Consequently, certain aspects of the physical world may need to be rescaled accordingly to achieve the desired effect.

Consider TCP, a protocol that depends on observed round-trip times to correctly compute retransmission timeouts. These timing measurements must be made in the dilated time frame. Because time dilation uniformly scales the passage of time, it not only increases perceived bandwidth, it also decreases perceptions of round-trip time. Thus, a network with 10-ms physical round trip time (RTT) would appear to have 1-ms RTT to dilated TCP. Because TCP performance is sensitive to RTT, such a configuration is likely undesirable. To address this effect, we independently scale bandwidth and RTT by using network emulation [89, 102] to deliver appropriate bandwidth and latency values. In this example, we increase link delay by a factor of 10 to emulate the jump in bandwidth-delay product one expects from the bandwidth increase.

Similar use cases exist for mechanisms to independently scale other resources such as the CPU and disk I/O characteristics. These are the subject matter of Chapter 5.

But such scaling is not possible in all situations. For instance, under time dilation, low level subsystems such as the memory I/O bus and the PCI bus will also appear faster. Thus, a memory benchmark such as `lmbench` [14] will report much shorter memory access latencies under time dilation. The vast majority of applications are impervious to such low level artifacts, since other latencies, such as network and disk I/O, typically dominate. However, applications that require preserving the real-world memory access latencies under a dilated time frame are currently unsupported.

Finally, hardware and software architectures may evolve in ways that time dilation cannot support. For instance, consider a future host architecture with TCP offload [81], where TCP processing largely takes place on the network interface rather than in the protocol stack running in the operating system. Our current implementation does not dilate time for firmware on network interfaces, and may not extend to other similar architectures. In particular, time dilation might not be able to extrapolate the behavior of a fundamentally different hardware architecture than the platform it is running on.

4.3.2 Timer Interrupts

The guest reads time values from Xen through a shared data structure. Xen updates this structure every time it delivers a timer interrupt to the guest. This happens on the following events:

1. when a domain is scheduled;
2. when the *currently executing* domain receives a periodic timer interrupt; and
3. when a guest-programmable timer expires.

We argued earlier that, for successful dilation, the number of timer interrupts delivered to a guest should be scaled down by the TDF. Of these three cases, we can only control the second and the third. By default our implementation does not change the scheduling pattern of the virtual machines for two reasons. First, we do not change the scheduling pattern because scheduling parameters are usually global and system wide. Scaling these parameters on a per-domain basis would likely break the semantics of many scheduling schemes. Second, we would like to retain scheduling as an independent variable in our system, and therefore not constrain it by the choice of TDF. Thus, users could choose any one of the three CPU schedulers available in Xen.

However, for applications that are sensitive to the exact scheduling patterns of a VM, it might be necessary to manipulate the scheduling pattern when running in a dilated time frame. Section 5.2 describes our modifications to the CPU scheduler to support such applications.

Also, reducing the frequency of interrupts to the currently executing domain has its side-effects. As the frequency of interrupts delivered goes down, the domain becomes more and more unresponsive, and thus unsuitable for interactive processing. However, as we shall show in the evaluation section, network emulation is still accurate up to a TDF of 100.

4.3.3 Uniformity: Outside the Dilation Envelope

Time dilation works as expected only when all components in a system are running at the same time dilation factor. In particular, any external interactions — such as network communication with remote hosts, or delays external to the VM — must be accounted for, otherwise the OS might experience an inconsistent state. Consider a single packet traversing the network. Packet processing within the VM occurs in the dilated time frame. Ideally, all packet processing external to the VM should also be uniformly dilated.

Specifically, we scale the time a packet spends inside the VM (since time is measured in the dilated frame) and the time a packet spends over the network (using traffic shaping tools or network emulators). However, we do not scale the time a packet spends inside the Xen hypervisor and Domain-0, or the time it takes to process the packet at the other end of the connection.

These unscaled components may affect the OS's interpretation of round trip time. Consider the time interval between a packet and its acknowledgment across a link of latency R scaled by S, and let δ denote the portion of this time that is unscaled. In a perfect world where everything is dilated uniformly, a dilated host would measure the interval to be simply $T_{perfect} = R + \delta$. A regular, undilated host measures the interval as $T_{normal} = S \times R + \delta$; a dilated host in our implementation would observe the same scaled by S, so $T_{dilated} = T_{normal}/S = (S \times R + \delta)/S$.

We are interested in the error relative to perfect dilation:

$$\begin{aligned}\epsilon &= \frac{(T_{perfect} - T_{dilated})}{T_{perfect}} \\ &= \left(1 - \frac{1}{S}\right)\left(\frac{\delta}{R + \delta}\right)\end{aligned}$$

Note that ϵ approaches $\delta/(R + \delta)$ when S is large. In the common case this is of little consequence. For the regime of network configurations we are most interested in (high bandwidth-delay product networks), the value of R is typically orders of magnitude higher than the value of δ. As our results in Section 4.4 show, dilation remains accurate over a wide range of round trip times, bandwidths, and time-dilation factors that we consider. To verify this, we empirically measured δ by collecting TCP RTT samples across a single Dummynet [89] hop. For each sample, we treat anything above the known two-way propagation delay of the link as comprising the unscaled time. Across multiple combinations of bandwidths and link delays (ranging from 10 Mbps/10 ms to 100 Mbps/100 ms), we measured this error to be less than 1 ms.

We next validate the expected behavior of time dilation and establish its accuracy.

4.4 Validation

We begin this section with a description of the general methodology used for validating time dilation. Next, we present a straw-man test to verify the expected behavior of time dilation

Table 4.2: Network scaling.

TDF	**Real Configuration**	**Perceived Configuration**
1	100 Mbps, 80 ms	100 Mbps, 80 ms
10	100 Mbps, 80 ms	1000 Mbps, 8 ms
10	10 Mbps, 800 ms	100 Mbps, 80 ms
t	B Mbps, L ms	B Mbps, L ms
t	B/t Mbps, $L \times t$ ms	B Mbps, L ms

by using older hardware running under a dilated time frame to predict the performance of newer, more capable hardware. We then validate time dilation across a variety of microbenchmarks. We compare the behavior of a single TCP flow subject to various network conditions under different time dilation factors. Finally, we evaluate time dilation in more complex settings with multiple flows.

4.4.1 Methodology

Observe that the goal of validation is to ensure that a system exhibits some expected behavior in a given, controlled environment. To define this "expected" behavior, we first establish a *baseline* performance using some readily available system. We then compare the performance of the test system running under time dilation with the baseline to determine the accuracy. Time dilation uniformly increases the perceived capacity in the test system by the scaling factor TDF. Hence, we configure test system such that after time dilation, the perceived resource capacity of the test system matches that of the baseline configuration (recall the notion of resource equivalence defined in the previous chapter). The baseline performance is usually defined by an application specific metric. Then the metric for accuracy under this resource equivalence condition is the performance differential between the test system and the baseline system.

For instance, our first experiment uses an older machine with a 500-MHz processor to predict the performance of a newer 2.6-GHz computer using a TDF of 5. In this case, the resource equivalence was possible because we had older, less powerful hardware at hand. But in general, it might not be possible or practical to construct the test system out of less capable hardware, otherwise we would need multiple distinct hardware ensembles to evaluate multiple scaling factors. We therefore need mechanisms to explicitly control the resource allocation in the dilated time frame such that resource equivalence is preserved. The next chapter describes the mechanisms for several different resources in great detail. For this chapter, we briefly describe the network scaling since our validation experiments here are focused primarily on the ability to accurately replicate network intensive workloads.

Consider two hosts connected to each other via a single network link. Assume further that the network characteristics — in particular the bandwidth and the latency — of this link are configurable. For the baseline, we configure the link to have 100-Mbps bandwidth and a one way latency of 40 ms (thus the RTT is 80 ms). In order to validate time dilation, we wish to observe the network performance under different time dilation factors, maintaining the invariant that the network characteristics, such as the link bandwidth and latency, as perceived by the end hosts remain unchanged.

But if the end hosts are run under a TDF of 10, they will perceive the RTT to be only 8 ms, since the 80-ms RTT in real time will only feel like 8-ms in the dilated time frame. Correspondingly, the end hosts will perceive a much higher bandwidth of 1000 Mbps. Clearly, this violates our desire to maintain resource equivalence. If instead the physical link is configured to have 10-Mbps bandwidth and a one way delay of 400 ms, then in the dilated time frame the link will appear to have 100-Mbps bandwidth and 80 ms RTT, which is precisely the desired outcome. In general, for a TDF of t, a link with bandwidth B and one way latency L should be reconfigured with bandwidth B/t and latency $L \times t$. Table 4.2 summarizes this discussion.

Existing traffic shaping tools can be leveraged at the end points of a given link for altering the network characteristics. Alternatively, since we already use network emulation environments such as Dummynet [89] and ModelNet [102] for our validation experiments, modifying link characteristics is a simple matter of reconfiguring the emulated topology with the appropriately scaled bandwidth and latency values.

4.4.2 Hardware Validation

We start by evaluating the predictive accuracy of time dilation using multiple generations of hardware. One of the key motivations for time dilation is as a predictive tool, such as predicting the performance and behavior of protocol and application implementations on future higher-performance network hardware. To validate time dilation's predictive accuracy, we use dilation on older hardware to predict TCP throughput as if we were using recent hardware. We then compare the predicted performance with the actual performance when using recent hardware.

We use time dilation on three hardware configurations, listed in Table 4.3, such that

Table 4.3: Validating performance prediction: the mean per-flow throughput and standard deviations of 50 TCP flows for different hardware configurations.

Configuration	**TDF**	**Mean (Mbps)**	**St.Dev. (Mbps)**
2.6 GHz, 1-Gbps NIC (restricted to 500-Mbps)	1	9.39	1.91
1.13 GHz, 1-Gbps NIC (restricted to 250-Mbps)	2	9.57	1.76
500 MHz, 100-Mbps NIC	5	9.70	2.04

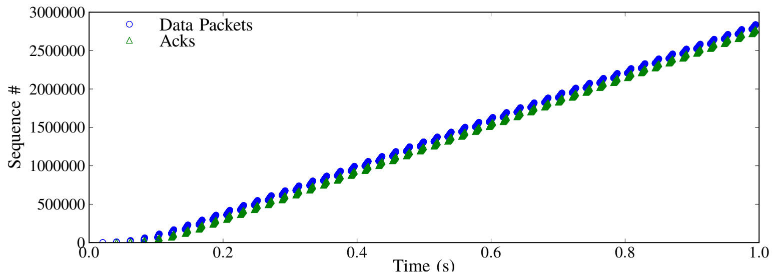
each configuration resembles a 2.5-GHz processor with a 500-Mbps NIC, under the coarse assumption that CPU performance roughly scales with processor frequency. The base hardware configurations are 500-MHz and 1.13-GHz Pentium III machines with 10/100-Mbps and 1-Gbps network interfaces, respectively, and a 2.6-GHz Pentium IV machine with a 1-Gbps network interface. In cases where the base hardware was not available (a 250-Mbps or a 500-Mbps NIC, for instance), we restrict the available bandwidth on the 1-Gbps interfaces using standard traffic shaping tools.

For each hardware configuration, we measure the TCP throughput of 50 flows communicating with another identical machine. Using Dummynet [89], we configure the network between the hosts to have an effective RTT of 80 ms. We then calculate the mean per-flow throughput and standard deviation across all flows. As Table 4.3 shows, both the mean and deviation of per-flow throughput are consistent across the hardware configurations, which span over an order of magnitude of difference in hardware performance. For example, time dilation using a 500 MHz CPU with a 100-Mbps NIC dilated with a TDF of 5 is able to accurately predict TCP throughput of a 2.6-GHz CPU with a 1-Gbps NIC (restricted to 500-Mbps using the Linux traffic shaping tools). For this admittedly simple example at least, we conclude that time dilation is an effective tool for making reasonable performance predictions for more capable hardware that what is actually available.

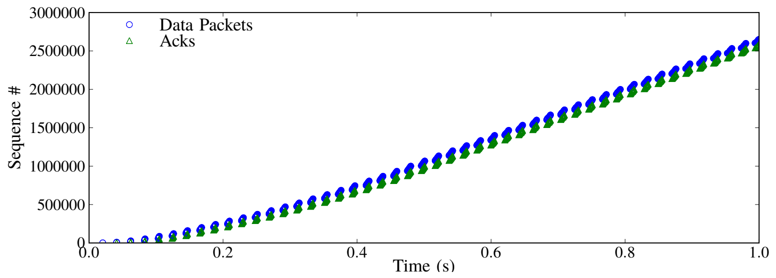
In the following experiments, we further examine the accuracy of time dilation on fine-grained packet level behavior.

4.4.3 Single Flow Packet-Level Behavior

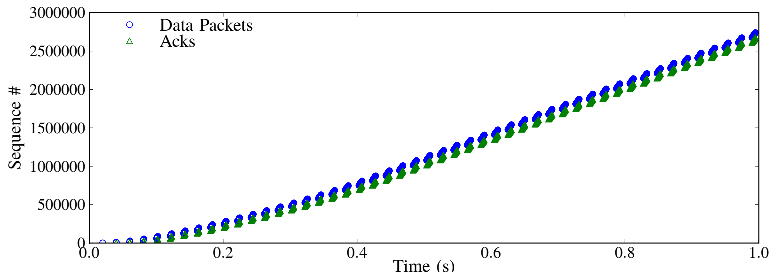
It is tempting to validate time dilation by comparing an easy to measure metric, such as average throughput of a TCP flow. However, such coarse metrics are fairly easy to reproduce, and do not capture many interesting properties that manifest themselves at finer-grained time intervals, such as the distribution of inter-packet arrival times. Hence, we next illustrate that



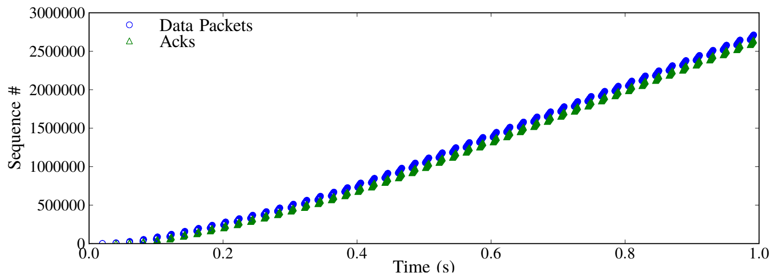
(a) Native Linux: link bandwidth 100 Mbps, link delay 10ms



(b) Xen VM (TDF 1): link bandwidth 100 Mbps, link delay 10 ms



(c) Xen VM (TDF 10): link bandwidth 10 Mbps, link delay 100 ms



(d) Xen VM (TDF 100): link bandwidth 1 Mbps, link delay 1000 ms

Figure 4.2: Packet timings for the first second of a TCP connection with no losses for native Linux and three time dilation configurations. In all cases, we configure link bandwidth and delay such that the bandwidth-delay product is constant.

time dilation preserves the perceived packet-level timing of TCP.

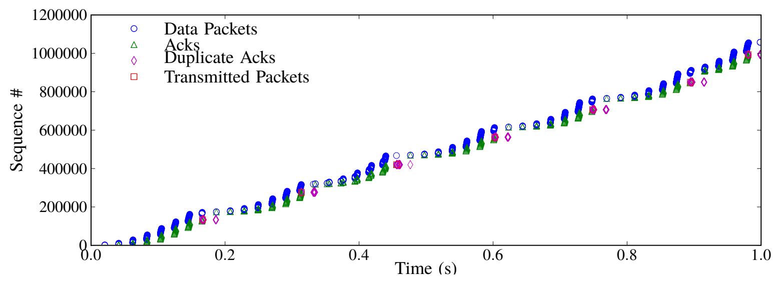
We use two end hosts directly connected through a Dell Powerconnect 5224 gigabit switch. Both systems are Dell PowerEdge 1750 servers with dual Intel 2.8-GHz Xeon processors, 1 GB of physical memory, and Broadcom NetXtreme BCM5704 integrated gigabit Ethernet NICs. The end hosts run Xen 2.0.7, modified to support time dilation. We use Linux 2.6.11 as the Xen guest operating system, and all experiments run inside of the Linux guest VMs. All protocols in our experiments use their default parameters unless otherwise specified. We use two identical machines running Linux 2.6.10 and Fedora Core 2 for our "unmodified Linux" results.

We control network characteristics such as bandwidth, delay, and loss between the two hosts using Dummynet. In addition to its random loss functionality, we extended Dummynet to support deterministic losses to produce repeatable and comparable loss behavior. Unless otherwise noted, all endpoints run with identical parameters (buffer sizes, TDFs, etc.).

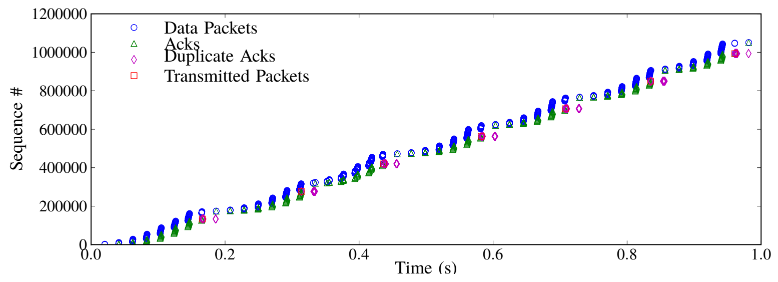
In this experiment, we first transfer data on TCP connections between two unmodified Linux hosts and use `tcpdump` [19] on the sending host to record packet behavior. We measure TCP behavior under both lossless and deterministic lossy conditions.

We then repeat the experiment with the sending host running with TDFs of {1, 10, 100}, spanning two orders of magnitude. When dilating time, we configure the underlying network such that a time-dilated host perceives the same network conditions as the original TCP experiment on unmodified hosts. For example, for the experiment with unmodified hosts, we set the bandwidth between the hosts to 100 Mbps and the delay to 10 ms. To preserve the same network conditions perceived by a host dilated by a factor of 10, we reduce the bandwidth to 10 Mbps and increase the delay to 100 ms using Dummynet. Thus, if we are successful, a time dilated host will see the same packet timing behavior as the unmodified case, even though the physical network conditions have changed substantially. We include results with TDF of 1 to compare TCP behavior in an unmodified host with the behavior of our system modified to support time dilation.

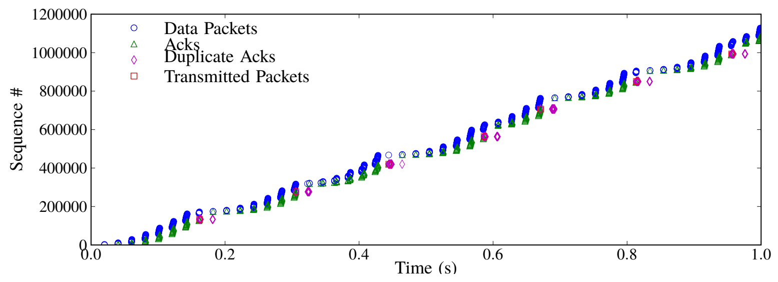
We show sets of four time sequence graphs in Figures 4.2 and 4.3. Each graph shows the packet-level timing behavior of TCP on four system configurations: unmodified Linux, and Linux running in Xen with our implementation of time dilation operating under TDFs of 1, 10, and 100. The first set of graphs shows the first second of a trace of TCP without loss. Each graph shows the data and ACK packet sequences over time, and illustrates TCP slow-start and transition to steady-state behavior. Qualitatively, the TCP flows across configurations have



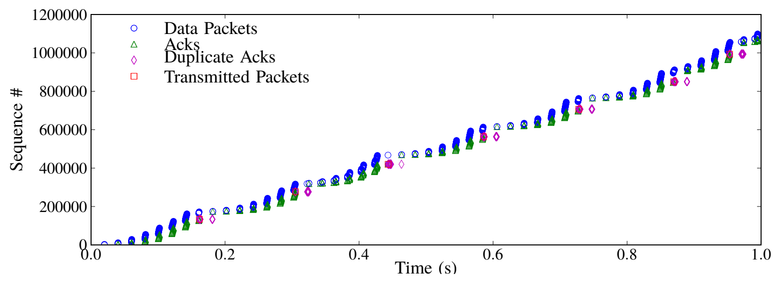
(a) Native Linux: link bandwidth 100 Mbps, link delay 10 ms



(b) TDF 1: link bandwidth 100 Mbps, link delay 10 ms

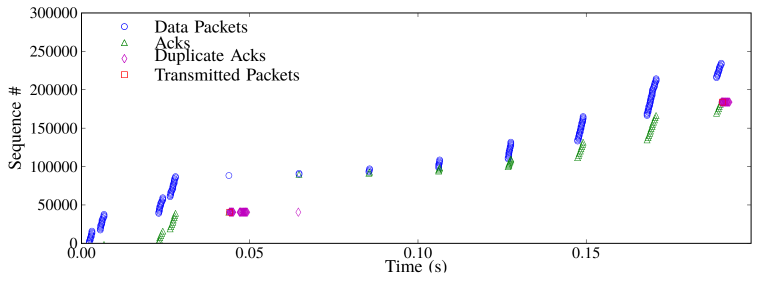


(c) TDF 10: link bandwidth 10 Mbps, link delay 100 ms

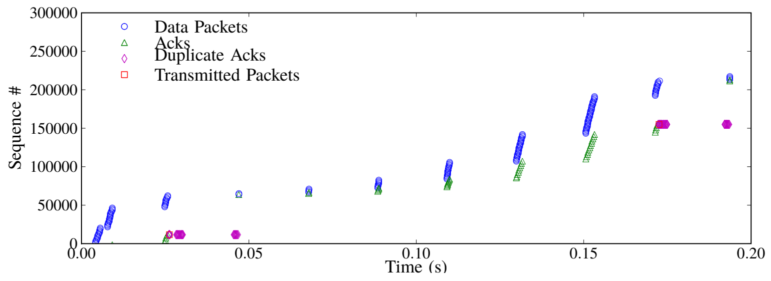


(d) TDF 100: link bandwidth 1 Mbps, link delay 1000 ms

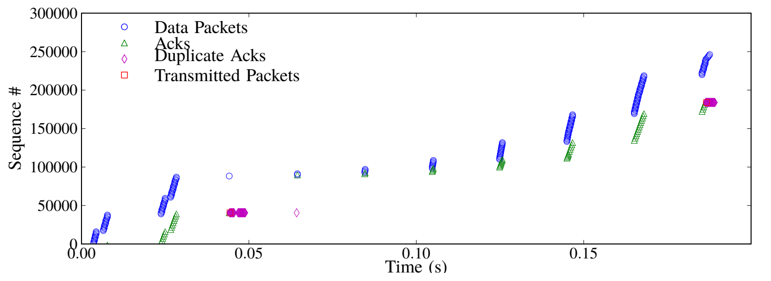
Figure 4.3: Packet timings for the first second of a TCP connection with 1% deterministic losses.



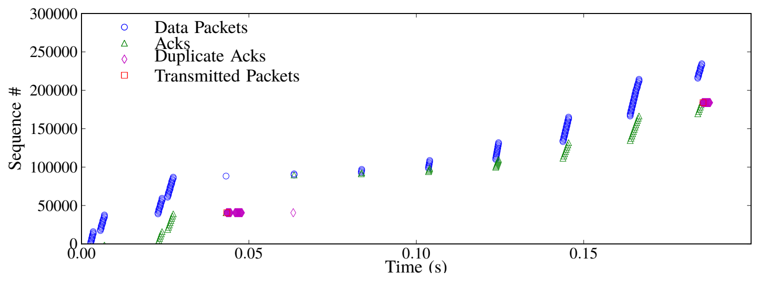
(a) Native Linux: link bandwidth 100 Mbps, link delay 10 ms



(b) TDF 1: link bandwidth 100 Mbps, link delay 10 ms

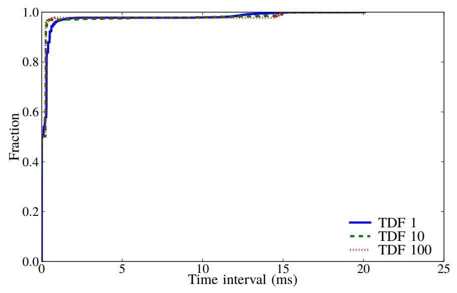


(c) TDF 10: link bandwidth 10 Mbps, link delay 100 ms

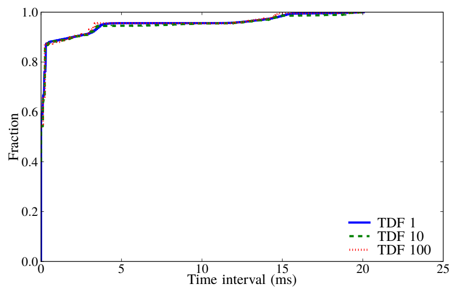


(d) TDF 100: link bandwidth 1 Mbps, link delay 1000 ms

Figure 4.4: Packet timings for 200 ms of the trace show in Figure 4.3 starting at an offset of 400 ms.



(a) Distribution under no loss



(b) Distribution under 1% loss

Figure 4.5: Comparison of inter-packet transmission times for a single TCP flow across 10 runs.

nearly identical behavior.

Comparing Figures 4.2a and 4.2b, we see that a dilated host has the same packet-level timing behavior as an unmodified host. More importantly, we see that time dilation accurately preserves packet-level timings perceived by the dilated host. Even though the configuration with a TDF of 100 has network conditions two orders of magnitude different from the base configuration, time dilation successfully preserves packet-level timings.

Time dilation also accurately preserves packet-level timings under lossy conditions. The second set of time sequence graphs in Figure 4.3 shows the first second of traces of TCP experiencing 1% loss. As with the lossless experiments, the TCP flows across configurations have nearly identical behavior even with orders of magnitude differences in network conditions. Finally, the third set of graphs in Figure 4.4 shows a 200 ms window of the same TCP traces starting at an offset of 400 ms into the trace. These graphs show a smaller time interval to illustrate the packet timings in more detail.

Figures 4.2 and 4.3 illustrate the accuracy of time dilation qualitatively. For a more quantitative analysis, we compared the distribution of the inter-arrival packet reception and transmission times for the dilated and undilated flows. Figure 4.5 plots the cumulative distribution function for inter-packet transmission times for a single TCP flow across 10 runs under both lossy and lossless conditions. Visually, the distributions track closely. Table 4.4 presents a statistical summary for these distributions, the mean and two indices of dispersion — the coefficient of variance (CoV) and the inter-quartile range (IQR) [62]. An index of dispersion indicates the variability in the given data set. Both CoV and IQR are unit-less, i.e., they take

Table 4.4: Statistical summary of inter-packet transmission times.

Metric	**No loss**			1% loss		
	TDF 1	TDF 10	TDF 100	TDF 1	TDF 10	TDF 100
Mean (ms)	0.458	0.451	0.448	0.912	1.002	0.896
CoV	0.242	0.218	0.206	0.298	0.304	0.301
IQR	0.294	0.248	0.239	0.202	0.238	0.238

the unit of measurement out of variability consideration. Therefore, the absolute values of these metrics is not of concern to us, but that they match under dilation is. Given the inherent variability in TCP, we find this similarity satisfactory. The results for inter-packet reception times are similar.

Having established the accuracy of time dilation in preserving low level packet behavior, we now present some examples of how time dilation can be used as an effective tool for pushing beyond hardware limitations to evaluate protocols and applications in resource rich environments. Section 5.1 presents further validation of time dilation in more complex scenarios in the presence of multiple VMs.

4.5 Applications

The ability of time dilation to create a realistic perception of more resources — faster CPUs, higher bandwidths and lower latencies — presents some unique opportunities for scalable network emulation. To refresh, time dilation enables researchers to extrapolate to scenarios that can not be supported by the underlying physical hardware, while preserving realism. Having performed micro-benchmarks to validate the accuracy of time dilation, we now demonstrate the utility of time dilation for two scenarios: network protocol evaluation and analyzing performance bottlenecks in high-bandwidth applications.

4.5.1 Protocol Evaluation

A key application of time dilation is as a tool for evaluating the behavior and performance of protocols and their implementations on future networks. As an initial demonstration of our system's utility in this space, we show how time dilation can support evaluating TCP variants designed for high bandwidth-delay network environments, in particular using the publicly available BiC [113] extension to the Linux TCP stack. BiC uses binary search to increase the congestion window size logarithmically — the rate of increase is higher when the current transmission rate is much less than the target rate, and slows down as it gets closer to the target rate.

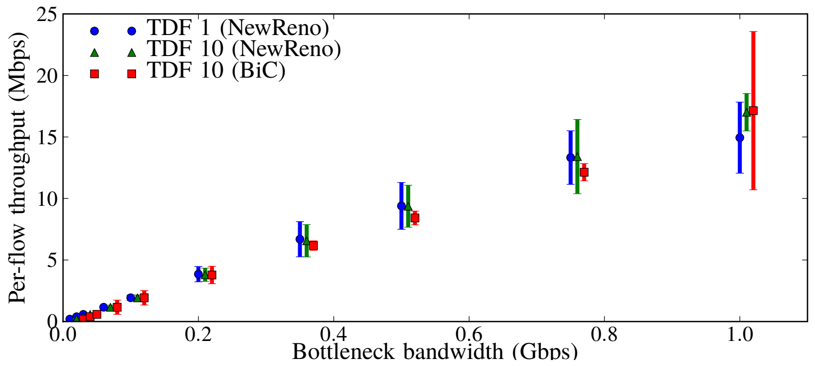
We set up a dumbbell topology with a single bottleneck link. We fix the RTT on the link to 80 ms, and vary the bandwidth. The experiment is run on machines connected with Gigabit Ethernet, and using time dilation we use the underlying 1-Gbps network to compare the default TCP flavor in Linux 2.6 kernels (NewRENO) with TCP BiC on a wide area link with bandwidth up to a 100 Gbps. We use Dummynet to manipulate the actual delay on the bottleneck link in order to preserve the 80 ms RTT under time dilation. We are limited from exploring higher bandwidths due to Dummynet's inability to accurately emulate extremely large delays. For example, 80 ms RTT at a TDF of 100 requires accurately emulating 4000 ms one way delay.

In this experiment, for a given network bandwidth we create 50 connections between two hosts with a lifetime of 1000 RTTs and measure the resulting throughput of each connection. We perform this experiment for the two TCP variants: NewRENO and BiC. In all of the following experiments, we adjust the Linux TCP buffers as suggested in the TCP Tuning Guide [13].

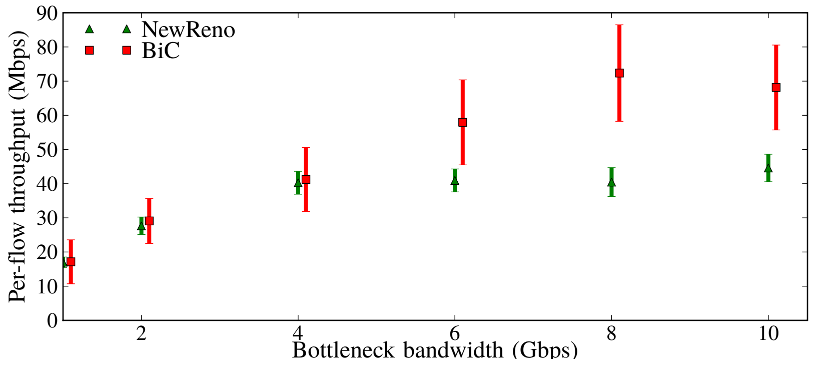
Figure 4.6 shows the per-flow throughput across the 50 connections as a function of network bandwidth. For each configuration, we plot the average per-flow throughput, and the error bar marks the standard deviation across all flows. In Figure 4.6a, the X-axis goes up to 1 Gbps, and represents the regime where the accuracy of time dilation can be validated against actual observations. Figures 4.6b (1 to 10 Gbps) and 4.6c (10 to 100 Gbps) show how time dilation can be used to extrapolate performance.

The graphs show three interesting results. First, time dilation enables us to experiment with protocols beyond hardware limits using implementations rather than simulations. Here we experiment with an unmodified TCP stack beyond the 1 Gbps hardware limit to 100 Gbps. Second, we can experimentally show the impact of high bandwidth-delay products on TCP implementations. Beyond 10 Gbps, per-flow TCP throughput starts to level off. Finally, we can experimentally demonstrate the benefits of new protocol implementations designed for such networks. Figure 4.6b shows that in the 1–10 Gbps regime, BiC outperforms TCP by a significant margin. However, in Figure 4.6c we see that TCP shows a steady, gradual improvement and both BiC and TCP level off beyond 10 Gbps.

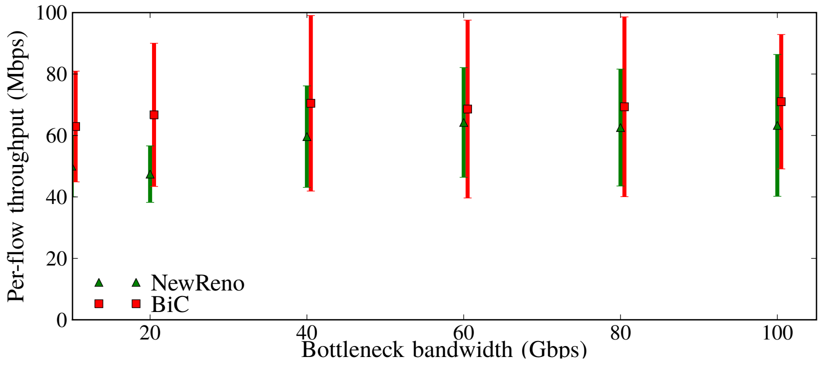
TCP performance is also sensitive to RTT. To show this effect under high-bandwidth conditions, we perform another experiment with 50 connections between two machines. How-



(a) Validating dilation: TCP performance under dilation matches actual, observed performance.



(b) Using dilation for protocol evaluation: comparing TCP with TCP BiC under high bandwidth.



(c) Pushing the dilation envelope: using a TDF of 100 to evaluate protocols under extremely high bandwidths.

Figure 4.6: Protocol Evaluation: Per-flow throughput of 50 flows for TCP and TCP BiC between two hosts on a network with an 80-ms RTT as a function of network bandwidth.

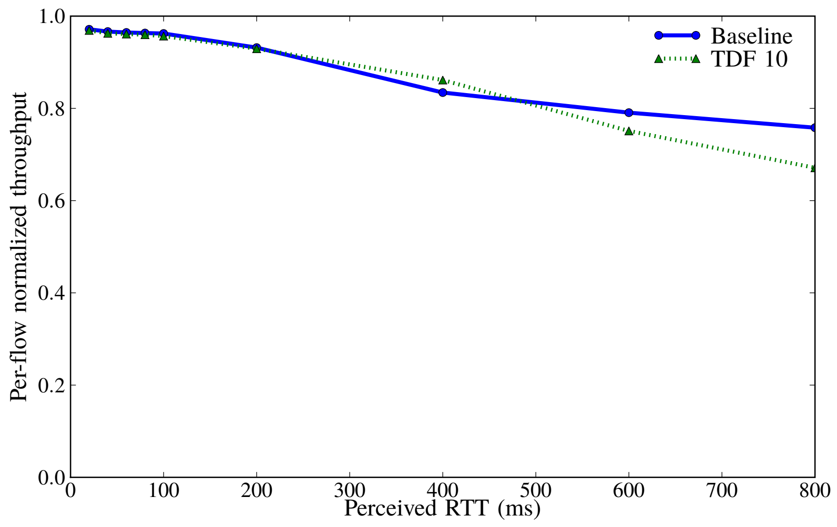


Figure 4.7: Protocol evaluation: Normalized average per-flow throughput of 50 flows for TCP and TCP BiC between two hosts on a network with 150-Mbps bandwidth as a function of RTT.

ever, this time we fix the network bandwidth at 150 Mbps and vary the perceived RTT between the hosts instead. For clarity, we present an alternative visualization of the results: instead of plotting the absolute per-flow throughput values, we instead plot normalized throughput values as a fraction of maximum potential throughput. For example, with 50 connections on a 150-Mbps bandwidth link, the maximum average per-flow throughput would be 3 Mbps. Our measured average per-flow throughput was 2.91 Mbps, resulting in a normalized per-flow throughput of 0.97.

Figure 4.7 shows the average per-flow throughput of TCP as a function of RTT from 0–800 ms. We chose this configuration to match a recent study on XCP [65], a protocol targeting high bandwidth-delay conditions. The results show the well-known dependence of TCP throughput on RTT, and that time dilation preserves this behavior.

4.5.2 High-bandwidth Applications

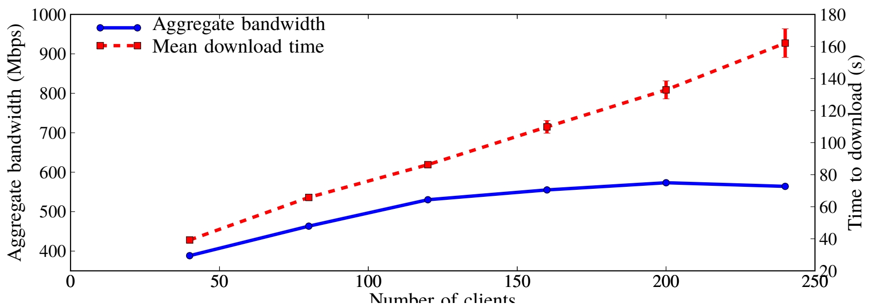
Time dilation can significantly enhance our ability to evaluate data-intensive applications with high bisection bandwidths using limited hardware resources. For instance, the recent popularity of peer-to-peer environments for content distribution and streaming requires significant aggregate bandwidth for realistic evaluations. Capturing the requirements of 10,000 hosts with an average of 1 Mbps per host would require 10 Gbps of emulation capacity and sufficient processing power for accurate study—a hardware configuration that would be prohibitively expensive to create for many research groups.

We show initial results of our ability to scale such experiments using modest hardware configurations with BitTorrent [7], a high-bandwidth peer-to-peer, content distribution protocol. Our goal is to explore the bottlenecks when running a large scale experiment using the publicly available BitTorrent implementation [7] (version 3.4.2).

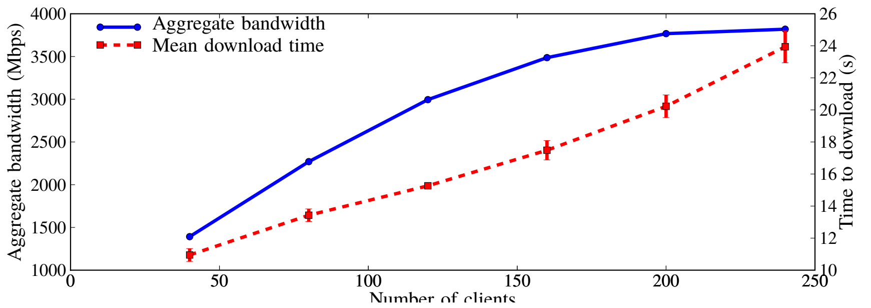
We conduct our experiments using 10 physical machines hosting VMs running BitTorrent clients interconnected through one ModelNet [102] core machine emulating an *unconstrained* network topology of 1,000 nodes. The client machines and the ModelNet core are physically connected via a Gigabit switch. The ModelNet topology is unconstrained in the sense that the network emulator forwards packets as fast as possible between endpoints. We create an overlay of BitTorrent clients, all of which are downloading a 46-MB file from an initial "seeder". We vary the number of clients participating in the overlay, distributing them uniformly among the 10 VMs. As a result, the aggregate bisection bandwidth of the BitTorrent overlay is limited by the emulation capacity of ModelNet, resource availability at the clients, and the capacity of the underlying hardware.

In the following experiments, we demonstrate how to use time dilation to evaluate BitTorrent performance beyond the physical resource limitations of the test-bed. As a basis, we measure a BitTorrent overlay running on the VMs with a TDF of 1 (no dilation). We scale the number of clients in the overlay from 40 to 240 (4–24 per VM). We measure the average time for downloading the file across all clients, as well as the aggregate bisection bandwidth of the overlay; we compute aggregate bandwidth as the number of clients times the average per-client bandwidth (file size/average download time). Figure 4.8a shows the mean and standard deviation for 10 runs of this experiment as a function of the number of clients. Since the VMs are not dilated, the aggregate bisection bandwidth cannot exceed the 1-Gbps limit of the physical network. From the graph, though, we see that the overlay does not reach this limit; with 200 clients or more, BitTorrent is able to sustain aggregate bandwidths only up to 570 Mbps. Increasing the number of clients further does not increase aggregate bandwidth because the host CPUs become saturated beyond 20 BitTorrent clients per machine.

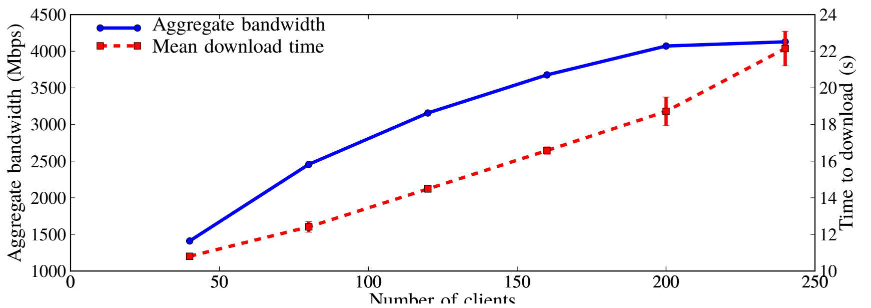
In the undilated configuration, CPU becomes a bottleneck before network capacity. Next we use time dilation to scale CPU resources to provide additional processing for the clients without changing the perceived network characteristics. To scale CPU resources, we



(a) VMs are running with TDF of 1 (no dilation). Performance degrades as clients contend for CPU resources.



(b) VMs are running with TDF of 10 and perceived network capacity is 1 Gbps. Dilation removes CPU contention, but network capacity becomes saturated with many clients.



(c) VMs are running with TDF of 10. Perceived network capacity is 10 Gbps. Increasing perceived network capacity removes network bottleneck, enabling aggregate bandwidth to scale until clients again contend for CPU.

Figure 4.8: Using time dilation for evaluating BitTorrent: Increasing the number of clients results in higher aggregate bandwidths, until the system reaches some bottleneck (CPU or network capacity). Time dilation can be used to push beyond these bottlenecks.

repeat the previous experiment but with VMs dilated with a TDF of 10. To keep the network capacity the same as before, we restrict the physical capacity of each client host link to 100 Mbps so that the underlying network appears as a 1-Gbps network to the dilated VMs. In effect, we dilate time to produce a new configuration with hosts with 10 times the CPU resources compared with the base configuration, interconnected by an equivalent network. Figure 4.8b shows the results of 10 runs of this experiment. With the increase in CPU resources for the clients, the BitTorrent overlay achieves close to the maximum 1-Gbps aggregate bisection bandwidth of the network. Note that the download times (in the dilated time frame) also improve as a result; due to dilation, though, the experiment takes longer in wall clock time (the most noticeable cost of dilation).

In the second configuration, network capacity now limits BitTorrent throughput. When using time dilation in the second configuration, we constrained the physical links to 100 Mbps so that the network had equivalent performance as the base configuration. In our last experiment, we increase *both* CPU resources and network capacity to scale the BitTorrent evaluation further. We continue dilating the VMs with a TDF of 10, but now remove the constraints on the network: client host physical links are 1 Gbps again, with a maximum aggregate bisection bandwidth of 10 Gbps in the dilated time frame. In effect, we dilate time to produce a configuration with 10 times the CPU and network resources as the base physical configuration.

Figure 4.8c shows the results of this last experiment. From these results, we see that the "faster" network leads to a significant decrease in download times (in the dilated time frame). Second, beyond 200 clients we see the aggregate bandwidth leveling out, indicating that we are again running into a bottleneck. On inspection, at that point we find that the end hosts are saturating their CPUs again as with the base configuration. Note, however, that in this case the peak bisection bandwidth exceeds 4 Gbps — performance that cannot be achieved natively with the given hardware.

We believe that time dilation is a valuable tool for evaluating large scale distributed systems by creating resource-rich environments. It represents a reasonable trade-off between realism and accuracy, compared to the current state-of-the-art. In fact, dilation enables scaling the size of the experiment as well. On the same set of 10 physical machines, we have successfully experimented with up to 400 BitTorrent clients.

DieCast further leverages time dilation to solve the converse problem, that of evaluating large distributed systems using a much smaller infrastructure. The next chapter discusses resource management mechanisms required by DieCast, in addition to time dilation.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2006. Gupta, Diwaker; Yocum, Kenneth; McNett, Marvin; Snoeren, Alex C.; Vahdat, Amin; Voelker, Geoffrey M. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Resource Scaling in a Dilated Time Frame

Section 3.1 introduced the notion of resource equivalence: a VM in a DieCast-scaled system should perceive the same resource capacity as its physical counterpart in the original system. Since multiplexing partitions the underlying resources among all the co-located VMs, we need mechanisms to restore fidelity and increase the perceived resource capacity of each VM to match its corresponding physical machine. In the previous chapter, we saw how time dilation can *uniformly* scale the perceived capacity of various system resources such as the CPU, network and disk I/O. However, time dilation alone is not sufficient to restore fidelity, especially in scenarios where the physical machines in the test harness differ significantly from the physical machines in the original system, or when there is non-trivial heterogeneity within machines in the original system.

Consider a system with two physical machines whose specifications appear in the first row of Table 5.1. Suppose we wish to use DieCast to replicate and test this system using a single test machine whose configuration appears in the second row of the same table. Note that the two machines in the original system have very different CPU, network and disk characteristics.

First, note that if we simply created two identical VMs on the test machine, each VM will get roughly half of the underlying resources. The resulting VMs perceive a drastically different environment than their physical counterparts. Next, if we run these (identical) VMs under a time dilation factor of two, then each VM perceives twice the resources that it actually has. As Table 5.1 shows (fourth row, last two columns), the first VM now perceives resources

Table 5.1: Restoring fidelity: Time dilation alone is not sufficient to match the resource characteristics of machines in the original system — mechanisms to independently scale various resources are required.

Original system			2.4-GHz CPU, 1-Gbps NW, Server-class HDD	2-GHz CPU, 100-Mbps NW, Desktop-class HDD
Test system	2.4-GHz CPU, 1-Gbps NW, Server-class HDD			
	Actual configuration		**Perceived configuration**	
Two identical VMs	1.2-GHz CPU, 0.5-Gbps NW, Server-class HDD	1.2-GHz CPU, 0.5-Gbps NW, Server-class HDD	1.2-GHz CPU, 0.5-Gbps NW, Server-class HDD	1.2-GHz CPU, 0.5-Gbps NW, Server-class HDD
Identical VMs @ TDF 2	1.2-GHz CPU, 0.5-Gbps NW, Server-class HDD	1.2-GHz CPU, 0.5-Gbps NW, Server-class HDD	2.4-GHz CPU, 1-Gbps NW, Server-class HDD	2.4-GHz CPU, 1-Gbps NW, Server-class HDD
Scaled VMs @ TDF 2	1.2-GHz CPU (50%), 0.5-Gbps NW (50%), Server-class HDD	1-GHz CPU (41.67%), 50-Mbps NW (5%), Desktop-class HDD	2.4-GHz CPU, 1-Gbps NW, Server-class HDD	2-GHz CPU, 100-Mbps NW, Desktop-class HDD

quite similar to that of the first physical machine. However, the second VM still perceives a very different configuration compared to the second physical machine in the original system. Thus, time dilation alone is clearly insufficient to restore fidelity in this case.

What we need, in fact, are mechanisms to precisely control the resources visible to individual VMs such that *after* time dilation, the perceived resource capacities match those of the machines in the original system. For instance, if the second VM is allocated only 41.67% of the underlying CPU, only 5% of the underlying network bandwidth, and if somehow the hard drive exposed to the second VM behaves like a desktop-class device rather than a server-class device, then after time dilation, the second VM will perceive a configuration very similar to the second physical machine. Note that we need independent knobs for each of the various resources in a system. These mechanisms are the subject of this chapter. We consider mechanisms for scaling three primary resources: the network, the CPU and the disk.

5.1 Network Scaling

Previous chapters have already covered some aspects of network scaling. In particular, we use traffic shaping tools and network emulators to configure the network topology such that perceived network characteristics such as per-hop bandwidths and latencies are preserved under time dilation. This section considers various facets of network scaling in more detail.

There are two aspects to scaling the network. First, there are the network resources on the local host itself. In particular, any end point has an inherent first-hop bottleneck — it cannot transmit or receive data at a rate higher than that supported by the physical network interface. Let us call appropriately scaling this host network interface *local network scaling*. Second, any network interactions external to the system must be scaled appropriately as well. In particular, the network properties of the original system must be preserved on a per-hop basis, as perceived by a time dilated system. We call this *external network scaling*.

Most modern network interface cards (NICs) support 1 Gbps bi-directional sustained throughput (also called duplex-mode). However, it is extremely rare for the bandwidth of the local interface to be the primary bottleneck in an end-to-end network path. It is much more likely that some intermediate, low-bandwidth link is the bottleneck. Hence, scaling the bandwidth at the local network interface is unlikely to have a measurable impact. Nevertheless, for completeness and correctness of the scaled system, we still throttle the local interface. Besides, there might be legitimate settings where the local interface *is* the primary bottleneck — on hosts connected with a 10-Gbps fabric, for instance.

Scaling the bandwidth on the local interface is quite trivial. Most modern operating systems already ship with traffic shaping tools that allow rate limiting traffic and traffic shaping on the local interface. In particular, Linux has a comprehensive suite of traffic shaping tools that we leverage to do the scaling [12]. Recall that Domain-0 is the privileged, control domain in Xen, and DieCast runs Linux as the guest OS within Domain-0. An artifact of the split device driver model in Xen is that, corresponding to each physical interface within a VM, Xen creates a virtual interface in Domain-0. We simply perform traffic shaping on the virtual interface(s) corresponding to the VM(s) in question. As before, if the local interface in the original system had a bandwidth B, then under time dilation with a TDF of t, we configure the interface to a maximum bandwidth of B/t.

Note that there is also a latency component attached to the local network interface. But unlike bandwidth, latency is additive. Which means that because the access latency for the local interface is substantially smaller than external network latencies, a situation where the local interface latency dominates is unlikely. Hence scaling the latency on the local interface is typically not required.

Of course, this discussion assumes that the traffic shaping tool of choice does its job faithfully. Control mechanisms tend to lose their accuracy at the extremes, when the bandwidths and latencies are too high or too low. Fortunately, time dilation is of significant value here. It may appear that the latencies involved in a local interface of the original system may be low enough that the additional software overhead associated with the traffic shaping tool could make it difficult to match the target latencies. To our advantage, maintaining accurate latency with time dilation actually requires *increasing* the real-time delay of a given packet; e.g., a 100-μs delay network link in the original network should be delayed by 1 ms when dilating by a factor of 10. Thus, network scaling actually simplifies the task of accurately emulating a larger/faster network environment.

We also note that we are able to use sophisticated traffic-shaping tools for DieCast because of the general-purpose environment in Domain-0. However, this does not preclude DieCast from operating on other virtualization platforms. For instance, the VMware ESX server ships with built-in mechanisms to shape traffic on a per-VM basis [104].

Next, we discuss external network scaling. Recall that the goal is to preserve the perceived network characteristics of each hop in the network topology. Ideally, if all components of a distributed system run under the same time dilation factor, including all the intermediate links, routers and switches, then the perceived network characteristics will be automatically preserved. For instance, each router will still honor the bandwidth constraints on flows (if any) *within* the dilated time frame. However, it might not be possible to port time dilation for such network appliances, due to limitations of the software or non-virtualizable hardware. Further, it is unclear how a physical link can support time dilation on its own — the link itself is simply a medium for transmitting data. Support for time dilation must come from either the end points or some other external mechanism.

One can always interpose an emulation environment to overlay an emulated topology on top of any given physical topology. This additional layer of indirection gives us a lot more flexibility in terms of deployment. If the original topology is being emulated in software using a network emulator, then doing network scaling is simply a matter of reconfiguring the bandwidth and latencies of each hop, as discussed in Section 4.4.1. However, just reconfiguring the per-hop characteristics does not take into account the queuing delays and the emulation overhead *within* the emulator.

Consider ModelNet [102], a software traffic shaper for the FreeBSD and Linux kernels. ModelNet maintains a series of queues corresponding to individual links in a target topology. It moves received packets from queue to queue in real time, appropriately delaying (or dropping) the packet according to the current hop-by-hop bandwidth, latency, and congestion characteristics of the target topology. Upon reaching its destination in the target topology, ModelNet forwards the packet to the appropriate VM in the test system. If the per-packet emulation time is non-trivial, then this time will be unaccounted for if we simply scale the per-hop characteristics of the original topology. An alternate approach would be to run the ModelNet core itself in a dilated time frame, and leaving the characteristics of the original network topology untouched. This approach would ensure that the emulator replicates the characteristics in the dilated time frame, which is precisely what we want. However, in most cases the overheads within the emulator are low enough that running it within a dilated time frame is not required. Such low overheads are useful if the emulator can not be virtualized, for instance.

DieCast relies on network emulation to do the external network scaling. We now present a more detailed validation of the network scaling mechanisms. To demonstrate that dilation preserves TCP behavior under a variety of conditions, even for short flows, we performed the following set of experiments under heterogeneous conditions. In these experiments, 60 flows shared a bottleneck link. We divided the flows into three groups of 20 flows, with each group subject to an RTT of 20 ms, 40 ms, or 60 ms. We also varied the bandwidth of the bottleneck link from 10 Mbps to 600 Mbps. We conducted the experiments for a range of flow lengths from 5 seconds to 60 seconds and verified that the results were consistent independent of flow duration.

We present data for one set of experiments where each flow lasts for 10 seconds. Figure 5.1 plots the mean and standard deviation across the flows within each group for TDFs of 1 (regular TCP) and 10. For all three groups, the results from dilation agree well with the undilated results: the throughputs for TDF of 1 match those for TDF of 10 within measured variability. Note that these results also reflect TCP's known throughput bias towards flows with smaller RTTs.

In our experiments thus far, all flows originated at a single VM and were destined to a single VM. However, when running multiple VMs (as might be the case to support, for instance, scalable network emulation experiments [102, 108]) one has to consider the impact of VMM scheduling overhead on performance. To explore the issue of VMM scheduling, we investigate the impact of spreading flows across multiple dilated virtual machines running on

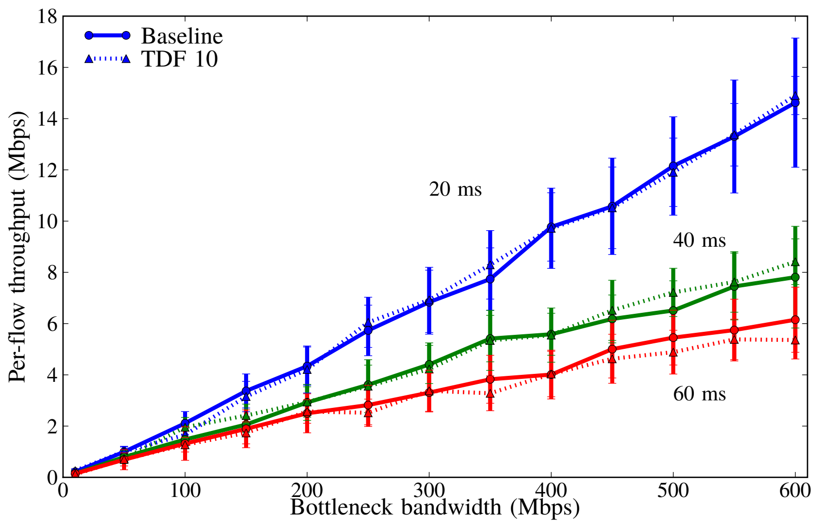


Figure 5.1: Per-flow throughput for 60 flows sharing a bottleneck link. Each flow lasts 10 seconds, and each group of 20 flows is subject to a different RTT. The mean and deviation are taken across the flows within each group.

the same physical host. In particular, we verify that simultaneously scheduling multiple dilated VMs does not negatively impact the accuracy of dilation.

In this experiment, for a given network bandwidth we create 50 connections between two hosts with a lifetime of 1000 RTTs and measure the resulting throughput of each connection. We configure the network with an 80 ms RTT, and vary the perceived network bandwidth from 0–4 Gbps using 1-Gbps switched Ethernet. Undilated TCP has a maximum network bandwidth of 1 Gbps, but time dilation enables us to explore performance beyond the raw hardware limits (we previously visited this point in Section 4.5.1). We repeat this experiment with the 50 flows split across 2, 5 and 10 virtual machines running on one physical machine.

Our results indicate that VMM scheduling does not significantly impact the accuracy of dilation. Figure 5.2 plots the mean throughput of the flows for each of the four configurations of flows divided among virtual machines. Error bars mark the standard deviation. Once again, the mean flow throughput for the various configurations are similar.

Thus, our mechanisms for network scaling remain accurate across a variety of network conditions and VM configurations. We discuss CPU scaling next.

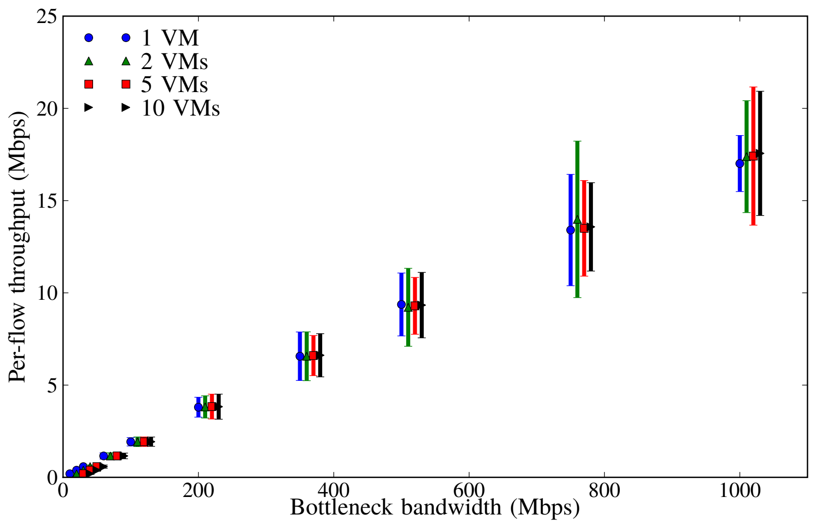


Figure 5.2: Mean throughput of 50 TCP flows between two hosts on a network with an 80-ms RTT as a function of network bandwidth. The 50 flows are partitioned among 1–10 virtual machines.

5.2 CPU Scaling

We begin this section with a brief discussion of the three CPU schedulers available in Xen. A good understanding of the features of these schedulers and the trade-offs they represent is important in deciding how to allocate CPU to VMs. Note that time dilation itself (and, hence, DieCast) is agnostic to the choice of CPU scheduler. We do, however, make modifications to CPU schedulers in some cases to improve the support for time dilation.

5.2.1 Three CPU Schedulers in Xen

Xen is unique among virtualization platforms in that it gives users a choice among different CPU schedulers, as a configurable boot-time parameter. However, this choice comes with the burden of choosing the right scheduler and configuring it appropriately. Since its inception, Xen has introduced three different CPU schedulers. We briefly characterize the main features of these schedulers below and motivate their inclusion in Xen. In the discussion below, we denote the work-conserving mode as WC-mode and the non-work-conserving mode as NWC-mode. For a description of these modes, please refer to Section 2.1.

Borrowed Virtual Time (BVT) [48] is a fair-share scheduler based on the concept of *virtual time*: the VM with the least virtual time is scheduled first. Additionally, BVT provides low-latency support for real-time and interactive applications by allowing latency-sensitive VMs to "warp" back in virtual time, thus gaining scheduling priority. The VM effectively borrows virtual time from its future CPU allocation.

The scheduler is configured with a context switch allowance C, which is the *real-time* by which the current VM is allowed to advance beyond another runnable VM with equal claim on the CPU. Each runnable domain receives a share of CPU in proportion to its weight $weight_i$. To achieve this, the virtual time of the currently running Dom_i is incremented by its running time divided by $weight_i$.

In summary, BVT has the following features:

- preemptive (if warp is used), WC-mode only;
- optimally-fair: the error between fair share and actual allocation is never greater than context switch allowance C plus one allocation slice;
- low-overhead implementation on multiprocessors as well as uni-processors.

The lack of NWC-mode in BVT severely limits its usage in a number of environments, and which brings us to this next scheduler.

Simple Earliest Deadline First (SEDF) [82] uses real-time algorithms to deliver hard guarantees on CPU allocation. Each domain Dom_i specifies its CPU requirements with a tuple (s_i, p_i, x_i), where the *slice* s_i and the *period* p_i together represent the CPU share that Dom_i requests: SEDF guarantees that Dom_i will receive at least s_i units of time in each period of length p_i. A boolean flag x_i indicates whether Dom_i is eligible to receive extra CPU if there is CPU slack in the system (WC-mode). SEDF distributes this slack time in a fair manner after all runnable domains have received their CPU share. One can allocate 30% CPU to a domain by specifying the requirement as either $(3\ ms, 10\ ms, 0)$ or $(30\ ms, 100\ ms, 0)$. The time granularity in the definition of the period impacts scheduler fairness.

In summary, SEDF has the following features:

- preemptive, supports both WC and NWC modes;
- fairness depends on the value of the period: shorter periods support allocation at finer time granularity;

- implements per-CPU queues: this design choice implies that SEDF lacks global load balancing on multiprocessors.

Credit Scheduler [8] is the most recent (and currently the default) CPU scheduler in Xen, featuring good out-of-the-box performance for average workloads and automatic load balancing of virtual CPUs across physical CPUs on a symmetric multi-processor (SMP) host. Before a CPU goes idle, it will consider other CPUs in order to find any runnable virtual CPU (VCPU). This approach guarantees that no physical CPU idles if there is any runnable VCPU in the system.

Each VM is assigned a *weight* and a *cap*. If the cap is 0, then the VM can receive any extra CPU (WC-mode). A non-zero cap limits the amount of CPU a VM receives (NWC-mode). The Credit scheduler uses 30 ms time slices for CPU allocation. A VM receives at most 30 ms of CPU time before being preempted to run another VM. Once every 30 ms, the priorities (credits) of all runnable VMs are recalculated. The scheduler monitors resource usage every 10 ms.

In summary, Credit has the following features:

- non-preemptive, supports both WC and NWC modes;
- global load balancing on multiprocessors.

In the course of this dissertation, we had occasion to use all of the above CPU schedulers at some point. In order to scale the CPU for DieCast, we want to precisely control the amount of CPU available to individual VMs. This is straightforward using the SEDF and Credit schedulers, since they directly support the NWC-mode. While the BVT scheduler does not directly support NWC-mode, a potential workaround is to run a CPU intensive job in a separate VM with the weights appropriately set such that the target VM can consume no more than the desired amount of CPU. Next, we evaluate the accuracy of CPU allocation for the SEDF and Credit schedulers.

5.2.2 Scheduler CPU Allocation Accuracy

A traditional metric used for analyzing and comparing CPU schedulers is the *error of CPU allocation*: it captures the difference between the specified CPU allocation and actual CPU consumption for a CPU intensive VM. To evaluate the CPU allocation error in NWC-mode for

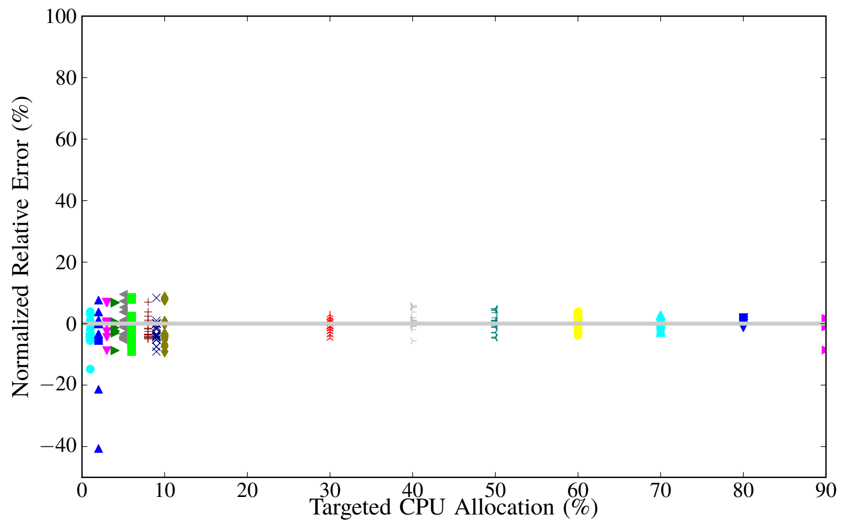


Figure 5.3: Allocation error in the SEDF scheduler.

SEDF and Credit schedulers, we designed a simple benchmark, called ALERT (**AL**location **ER**ror **T**est):

- Domain-0 is allocated a fixed, 6% CPU share during the all benchmark experiments;
- the guest domain Dom_1 executes a CPU-intensive task;
- for each benchmark point i the CPU allocation to Dom_1 is fixed to A_i, and each experiment i continues for 3 min;
- the experiments are performed with A_i = 1%, 2%, 3%, ..., 10%, 20%, ... , 90%.

This benchmark minimizes contention for resources: there is always enough CPU for Dom_1 to receive its allocated CPU. While ALERT does not guarantee the same CPU allocation accuracy when there are competing VMs, one can easily extend ALERT to test the CPU allocation error for multiple VMs, as well as for multi-VCPU VMs. The goal of ALERT is to establish the baseline accuracy of CPU allocation by designing the simplest, minimal experiment.

Let U_i^k denote CPU usage of Dom_1 measured during the k-th time interval in benchmark experiment i. If a CPU scheduler works accurately we should see that for $X\%$ of CPU allocation to Dom_1 it should consume X % of CPU, i.e., ideally, $U_i^k = A_i$ for any k-th time

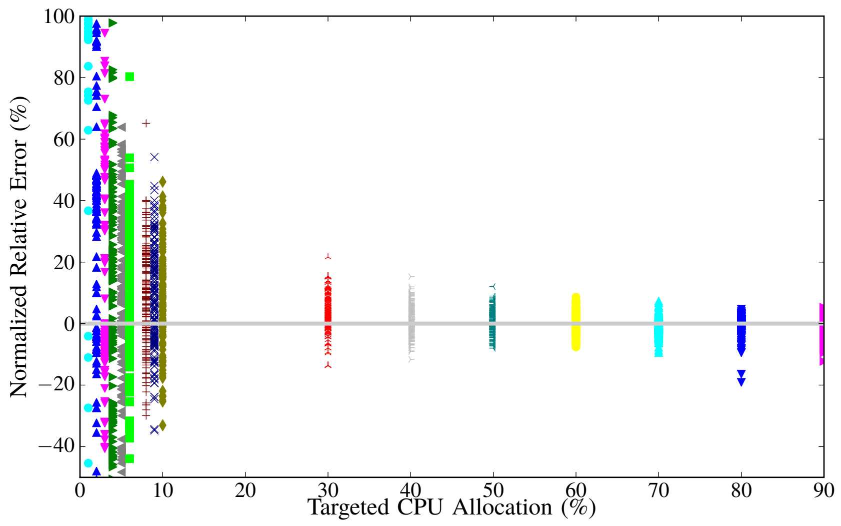


Figure 5.4: Allocation error in the Credit scheduler.

interval in benchmark experiment i. Let Er_i^k denote the *normalized relative error of CPU allocation*:

$$Er_i^k = (A_i - U_i^k)/A_i$$

We execute the ALERT benchmark under the SEDF and Credit schedulers in Xen. The Credit scheduler uses 30 ms as a time slice for CPU allocation as described in Section 5.2.1. To match the CPU allocation time granularity we use 10 ms period in SEDF in our comparison experiments.

Figures 5.3 and 5.4 show the normalized relative errors of CPU allocation with ALERT for SEDF and Credit schedulers respectively at one second time granularity, i.e., we compare the CPU usage U_i^k of Dom_1 measured at each second during experiment i with the assigned CPU allocation value A_i. X-axes represent the targeted CPU allocation, Y-axes show the normalized relative error. Each experiment is represented by 180 measurements (3 min = 180 sec); thus, each “stack” in Figures 5.3 and 5.4 has 180 points and the stack’s density reflects the error distribution.

As Figure 5.3 shows, the CPU allocation errors under SEDF are consistent and relatively small across all of the tested CPU allocation values. The Credit scheduler has much higher allocation error overall as shown in Figure 5.4. The errors are especially high for

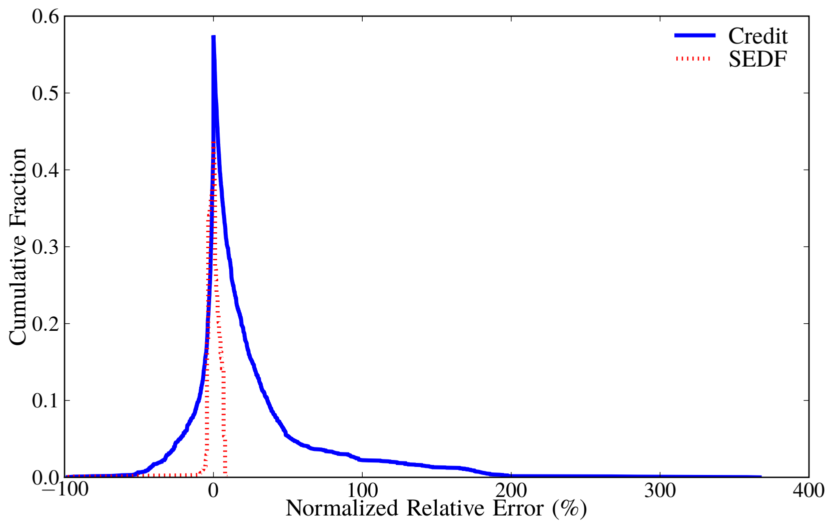


Figure 5.5: CDF_{-}^{+} of CPU allocation errors.

smaller CPU allocation targets, say below 30%. [1]

Figure 5.5 presents the relative distribution of all the errors measured during the ALERT experiments for the SEDF and Credit schedulers. We plot the normalized relative errors measured at 1-second time scale for all the performed experiments in ALERT ($18 \times 180 = 3240$ data points). The figure is a combination of the complementary CDFs of the negative errors as well as the positive errors. We call it CDF_{-}^{+}. From the figure, we can see that the Credit scheduler is over-allocating the CPU share more often than under-allocating (note the area under the curve for the positive errors), while for SEDF under-allocation is more common. As apparent from Figure 5.5 the Credit scheduler has a much higher CPU allocation error compared to SEDF scheduler.

Figure 5.6 shows the normalized relative errors of CPU allocation at three minute time scale, i.e., we compare the targeted CPU allocation with average CPU utilization measured at the end of each ALERT experiment (each experiment runs for 3 min). Overall, SEDF and Credit show comparable CPU allocation averages over longer time scale. However, the Credit scheduler's errors are still significantly higher than SEDF's errors for CPU allocation in the range [1%, 30%] as shown in Figure 5.6.

[1] We had to limit the shown error in Figure 5.4 to the range of $[-50\%, 100\%]$ for visibility: the actual range of the observed errors is $[-100\%, 370\%]$.

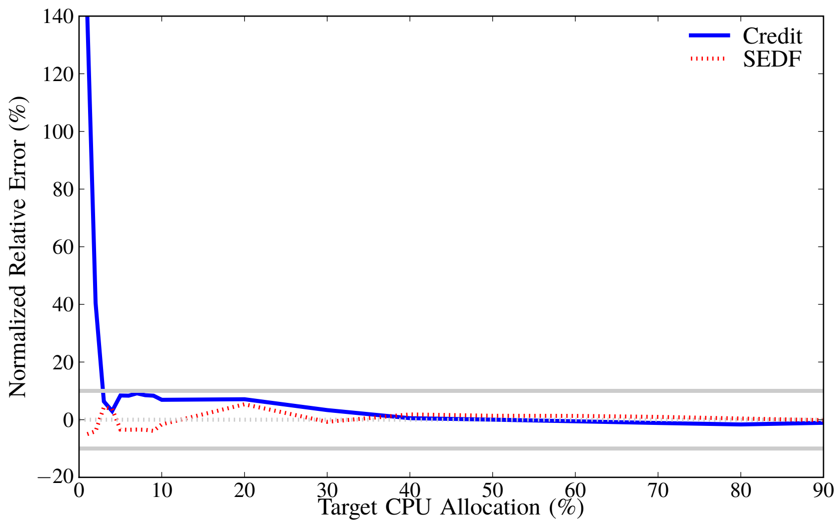


Figure 5.6: Normalized relative error over three minute intervals.

Allocation accuracy on multi-processors. To efficiently use CPU resources in SMP machines, special support is required from the underlying CPU scheduler. Recall that unlike the Credit scheduler, neither BVT nor SEDF support global load balancing, which limits their CPU usage efficiency. In other words, only the Credit scheduler has mechanisms to transparently migrate VCPUs to idle physical CPUs in the system.

To evaluate the accuracy of the Credit scheduler on SMP machines, we setup three VMs on a dual-CPU machine, with we allocate one-third (66%) of the *total* CPU resources to each VM. As before, each VM is hosting a CPU-intensive task. Each experiment lasts ten minutes and the CPU allocation error is measured at one second intervals. Figure 5.7 presents the CDF^{+}_{-} for this experiment. As the figure shows, the CPU allocation error introduced by the global load balancing scheme is relatively high when observed at a fine time granularity.

CPU Scaling in DieCast. DieCast employs a non-work conserving scheduler to ensure that each virtual machine receives no more than its allotted share of resources even when spare capacity is available. Suppose a CPU intensive task takes 100 seconds to finish on the original machine. The same task would now take 1000 seconds (of real time) on a dilated VM, since it can only use a tenth of the CPU. However, since the VM is running under time dilation, it only perceives that 100 seconds have passed. Thus, the VMs show similar behavior in terms

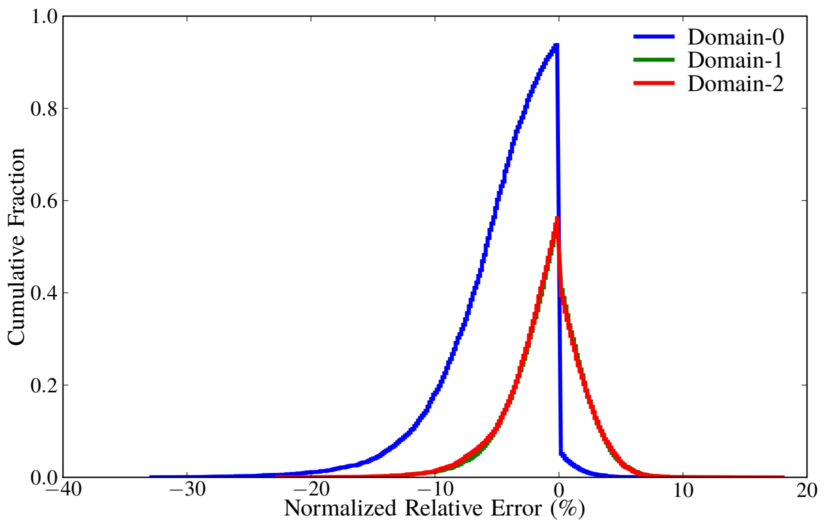


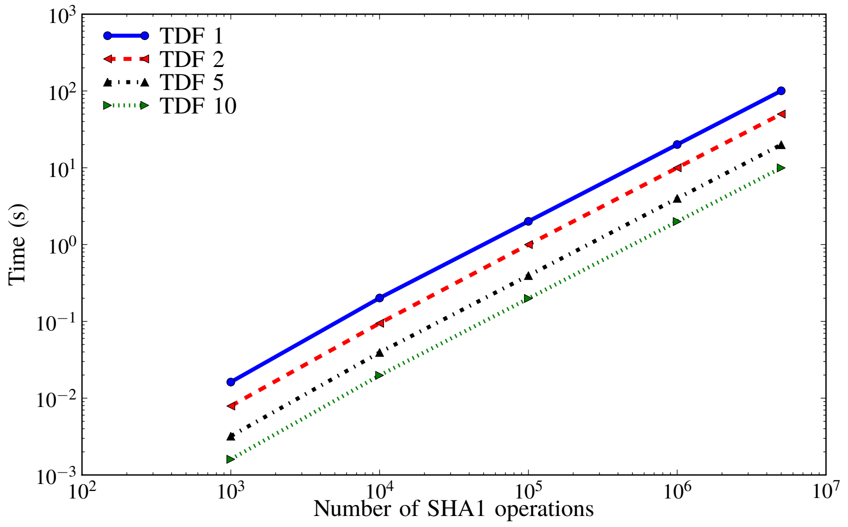
Figure 5.7: CPU allocation error, Credit, SMP case, 3 domains, NWC-mode.

of perceived processing power.

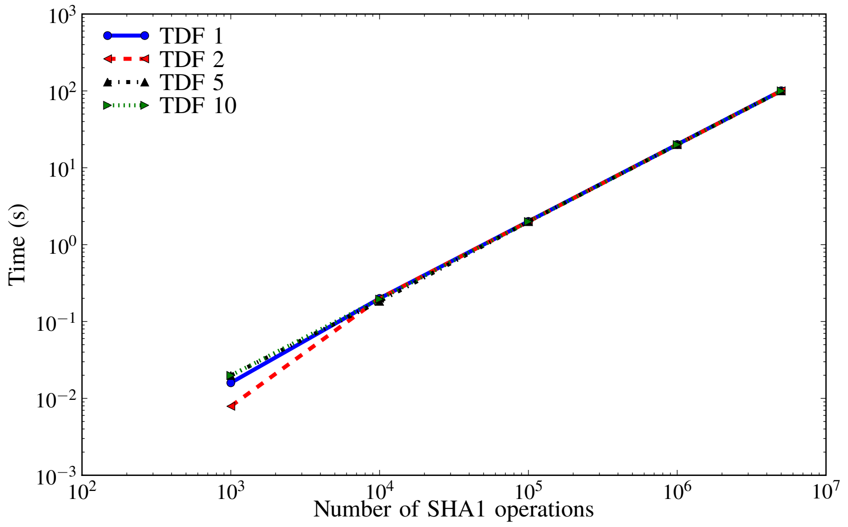
Unless otherwise stated, we use Xen's Credit scheduler to allocate an appropriate fraction of CPU resources to each VM in NWC-mode. However, simply scaling the CPU does not govern how those CPU cycles are distributed across time. Thus, a VM that is allocated 10% CPU might receive its 10% share in a single burst of a long, uninterrupted execution, or the 10% might get spread out across multiple, smaller executions.

The original Credit scheduler can schedule the same VM in consecutive time slices, as long as it does not consume its allocated share of the CPU. Thus, if a VM set to be dilated by a factor of 10 consumes less than 10% of the CPU in each time slice, then it can potentially get scheduled on *every* invocation of the scheduler, since in aggregate it never consumes more than its share of 10% CPU. This potential to run continuously distorts the performance of I/O-bound applications under dilation. In particular, they will observe a different timing distribution than they would in the real time frame. This distortion increases with increasing TDF. We found that, for some workloads, we actually need to enforce that a VM's CPU allocation is spread more *uniformly* across multiple executions.

We modify the Credit CPU scheduler in Xen to support this mode of operation as follows: if a VM runs for the entire duration of its time slice, we ensure that it does *not* get scheduled for the next $(tdf - 1)$ time slices. If a VM voluntarily yields the CPU or is



(a) The default behavior under time dilation is to uniformly increase the perceived processor power. Thus the same computation takes lesser and lesser time within the dilated time frame.



(b) Under a TDF of f, a VM with $1/f$ of the underlying CPU behaves like an undilated machine consuming 100% CPU.

Figure 5.8: CPU scaling.

pre-empted before its time slice expires, it *may* be re-scheduled in a subsequent time slice. However, as soon as it consumes a cumulative total of a time slice's worth of run time (carried over from the previous time it was descheduled), it will be pre-empted and not allowed to run for another $(tdf - 1)$ time slices.

5.2.3 Validation

Our first set of experiments validates the following claim. In a simple model, a VM with TDF of 10 running with 10% of the CPU has the same per-packet cycle budget as an undilated VM running with 100% of the CPU. Conversely, a VM running under a TDF of 10 perceives a ten times faster CPU, in terms of the available processing power (cycles/second) and consequently, in terms of the time taken to accomplish a given CPU intensive task.

Figure 5.8 shows the results of an experiment to validate this hypothesis. The experiment involves a single VM performing some number of SHA-1 hashes. This computation is CPU intensive, and by varying the number of hashes to compute, we can control the time it takes to finish the benchmark. Figure 5.8a demonstrates the default behavior under time dilation — as the dilation factor increases, the time taken to perform a given task (as measured within the VM) decreases. We then repeat the same experiment, except this time the CPU is scaled such that after time dilation, the perceived CPU capacity remains unchanged. Thus, for a TDF of 10, the VM is restricted to 10% of the underlying CPU. We use the Credit scheduler to specify the CPU reservations. As Figure 5.8b shows, after CPU scaling, the time taken for the computations to finish under various time dilation factors matches that of the baseline configuration without time dilation.

Such CPU scaling also allows us to control the perceived CPU capacity independent of the network characteristics. We validate this hypothesis by running an experiment similar to that described for Figure 5.2. This time, however, we adjust the VMM's CPU scheduling algorithm to restrict the amount of CPU allocated to each VM. We use the BVT scheduler to assign appropriate weights to each domain, and a CPU intensive job in a separate domain to consume the surplus CPU. Note that schedulers with native support for NWC-mode — such as Credit and SEDF — can simply be configured with appropriate CPU shares.

The goal of this experiment is to show that CPU scaling allows faithful emulation of CPU-bound scenarios. First, we find an undilated scenario that is CPU-limited by increasing link capacity. Because the undilated processor has enough power to run the network at line speed, we reduce its CPU capacity by 50%. We compare this to a VM dilated by TDF of

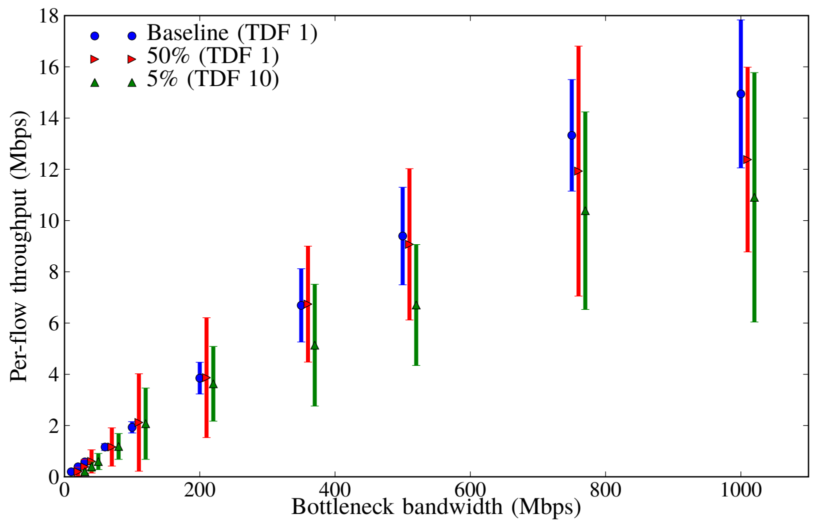


Figure 5.9: Per-flow throughput of 50 TCP flows between a CPU-scaled sender and unconstrained receiver. CPU utilization at the sender is restricted to the indicated percentages.

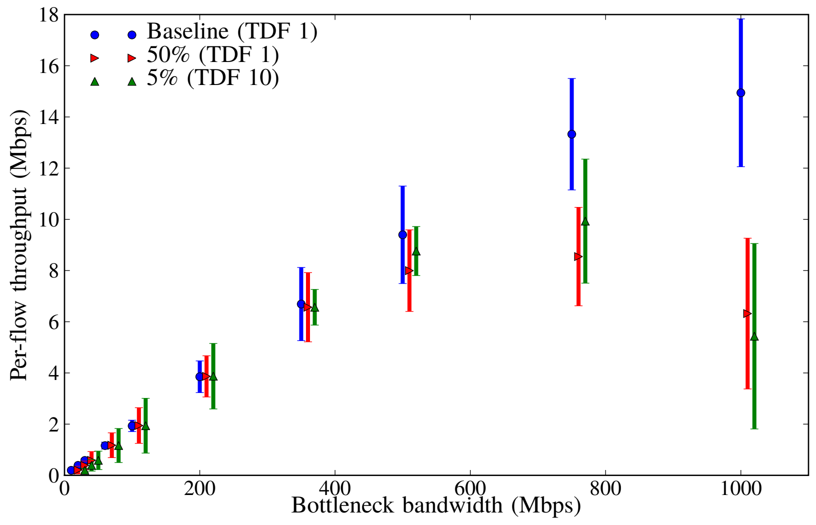


Figure 5.10: Per-flow throughput of 50 TCP flows between an unconstrained sender and a CPU-scaled receiver. CPU utilization at the receiver is restricted to the indicated percentages.

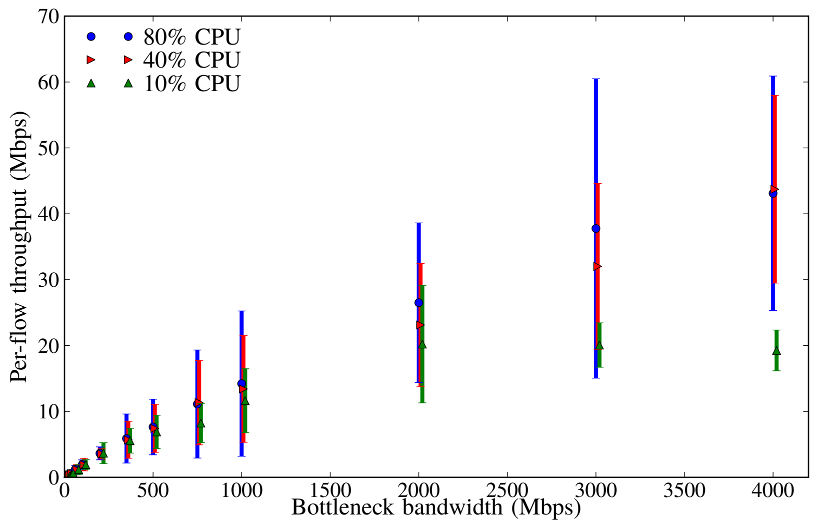
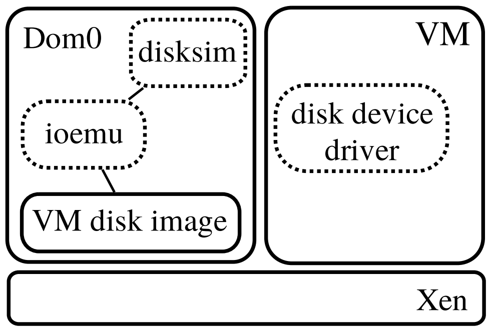


Figure 5.11: Per-flow throughput of 50 TCP flows across two hosts as a function of network bandwidths. CPU utilization at the sender is restricted to the indicated percentages. Experiments run with TDF of 10.

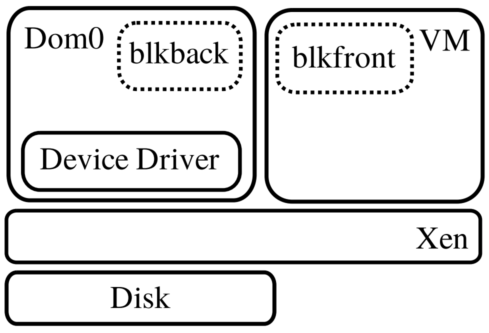
10 whose CPU has been scaled to 5%. The experimental setup is identical to that in Figure 5.2: 50 flows, 80-ms RTT. For clarity, we first throttled the sender alone, leaving the CPU unconstrained at the receiver; we then repeat the experiment with the receiver alone throttled. Figures 5.9 and 5.10 show the results. We plot the per-flow throughput, and error bars mark the standard deviation.

If we successfully scale the CPU, flows across a dilated link of the same throughput will encounter identical CPU limitations. Both figures confirm the effectiveness of CPU scaling, as the 50% and 5% lines match closely. The unscaled line (100%) illustrates the performance in a CPU-rich environment. Moreover our system accurately predicts that receiver CPU utilizations are higher than the sender's, confirming that it is possible to dilate CPU and network independently by leveraging the VMM CPU schedulers.

We can also use time dilation as a tool to estimate the computational power required to sustain a target bandwidth. For instance, from Figure 5.11, we can see that across a 4-Gbps pipe with an 80-ms RTT, 40% CPU on the sender is sufficient for TCP to reach around 50% utilization. This means that processors that are four times as fast as today's processors will be needed to achieve similar performance (since 40% CPU at TDF of 10 translates to 400%



(a) I/O Model for fully-virtualized VMs



(b) I/O Model for para-virtualized VMs

Figure 5.12: For para-virtualized VMs, we inject per-request delays in the `blkfront` device driver. Disk scaling for fully-virtualized VMs is delegated to a per-VM `disksim` process that integrates with `ioemu` to appropriately delay each request.

CPU at TDF of 1).

5.3 Disk Scaling

As with other resources that we have considered thus far, the default behavior under time dilation is that a VM will perceive a much faster disk with a higher throughput and a lower response time. As the example in beginning of this chapter demonstrated, DieCast requires the ability to precisely control the disk characteristics perceived by a VM. In other words, we would like to specify the disk characteristics that are exposed on a per-VM basis.

Appropriately scaling disk performance is a research challenge in its own right. Disk performance depends on such factors as head and track-switch time, SCSI-bus overhead, controller overhead, and rotational latency. A simple starting point would be to vary disk performance assuming a linear scaling model, but this could potentially violate physical properties inherent in disk drive mechanics. Besides, the test harness might be equipped with hard drives that differ significantly from the hard drives in the physical machines on the original system. Thus, DieCast effective requires some mechanism to decouple the actual physical hard drive from the disk that is exposed to individual VMs.

Ideally, we would scale individual disk requests at the disk controller layer. The complexity of modern drive architectures, particularly the fact that much low level functionality is implemented in firmware, makes such implementations challenging. Note that simply delaying requests in the device driver is not sufficient, since disk controllers may re-order and batch requests for efficiency. On the other hand, functionality embedded in hardware or firmware is difficult to instrument and modify. Further complicating matters are the different I/O models in Xen: one for para-virtualized VMs and one for fully-virtualized VMs. DieCast provides mechanisms to scale disk I/O for both models.

For fully-virtualized VMs, DieCast integrates a highly accurate and efficient disk system simulator — DiskSim [58] — which gives us a good trade-off between realism and accuracy. Figure 5.12a depicts our integration of DiskSim into the fully virtualized I/O model: for each VM, a dedicated user-space process (`ioemu`) in Domain-0 performs I/O emulation by exposing a "virtual disk" to the VM (the guest OS is unaware that a real disk is not present). A special file in Domain-0 serves as the backend storage for the VM's disk. To allow `ioemu` to interact with DiskSim, we wrote a wrapper around the simulator for inter-process communication.

After servicing each request (but before returning), `ioemu` forwards the request to DiskSim, which then returns the time, rt, the request would have taken in its simulated disk. Since we are effectively layering a software disk on top of `ioemu`, each request should ideally take exactly time rt in the VM's time frame, or $tdf * rt$ in real time. If $delay$ is the amount by which this request is delayed, the total time spent in `ioemu` becomes $delay + dt + st$, where st is the time taken to actually serve the request (DiskSim only simulates I/O characteristics, it does not deal with the actual disk content) and dt is the time taken to invoke DiskSim itself. The required delay is then $(tdf * rt) - dt - st$. Since all the quantities here are either known or easily measured, computing the required delay is straightforward.

The architecture of DiskSim, however, is not amenable to integration with the para-virtualized I/O model (Figure 5.12b). In this "split I/O" model, a front-end driver in the VM (`blkfront`) forwards requests to a back-end driver in Domain-0 (`blkback`), which are then serviced by the real disk device driver. Thus para-virtualized I/O is largely a kernel activity, while DiskSim runs entirely in user-space. Further, a separate DiskSim process would be required for each simulated disk, whereas there is a single back-end driver for all VMs.

For these reasons, for para-virtualized VMs, we inject the appropriate delays in the `blkfront` driver. This approach has the additional advantage of containing the side effects of such delays to individual VMs — `blkback` can continue processing other requests as usual. Further, it eliminates the need to modify disk-specific drivers in Domain-0. We emphasize that this design is functionally equivalent to per-request scaling in DiskSim: the key difference is that scaling in DiskSim is much closer to the (simulated) hardware. Overall our implementation

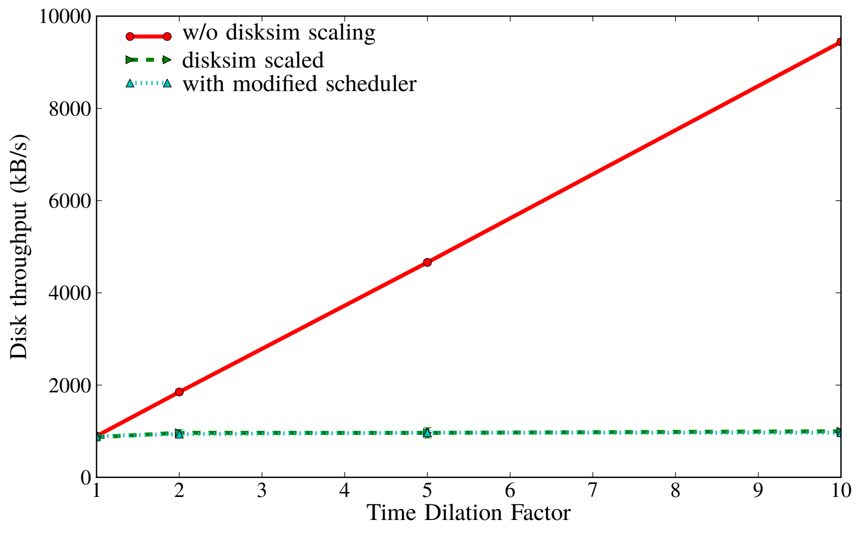


Figure 5.13: dd throughput under time dilation using DiskSim.

of disk scaling for para-virtualized VM's is simpler though less accurate and somewhat less flexible since it requires the disk subsystem in the testing hardware to match the configuration in the target system.

We have validated both our implementations using several micro-benchmarks. First, we run dd, a Unix tool for performing raw I/O to disk. Figure 5.13 shows the read throughput from a disk under various time dilation factors. The solid red line shows that in the absence of any disk I/O scaling, the read throughput exhibits a near-linear increase. Note that this baseline configuration also runs DiskSim, except that no per-request scaling takes place. With the per-request scaling turned on, we see that the read performance remains constant even under increasing dilation factors.

But dd is an admittedly simple, albeit powerful tool. We next run DBench [98] — a popular hard-drive and file-system benchmark — under different dilation factors and plot the reported throughput. Figure 5.14 shows the results for the fully-virtualized I/O model with DiskSim integration (results for the para-virtualized implementation can be found in a separate technical report [55]). Ideally, the throughput should remain constant as a function of the dilation factor. We first run the benchmark without scaling disk I/O or CPU, and we can see that the reported throughput increases almost linearly, an undesirable behavior. Next, we repeat the experiment and scale the CPU alone (thus, at TDF 10 the VM only receives 10%

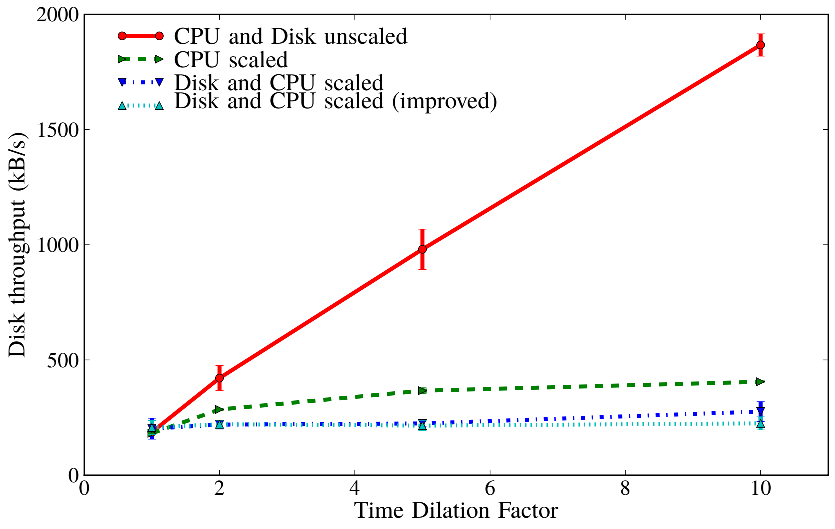


Figure 5.14: DBench throughput under time dilation using DiskSim.

of the CPU). While the increase is no longer linear, in the absence of disk dilation it is still significantly higher than the expected value. Finally, with disk dilation in place we can see that the throughput closely tracks the expected value.

However, as the TDF increases, we start to see some divergence. After further investigation, we found that this deviation results from the way we scaled the CPU. In particular, this is an example of a workload where the distribution, and not just the aggregate fraction, of CPU cycles becomes important. Recall our modifications to the Credit scheduler described in Section 5.2. The final line in figure 5.14 shows the results of the DBench benchmark with using this modified scheduler. As we can see, the throughput remains consistent even at higher TDFs. Note that unlike in this benchmark, DieCast typically runs multiple VMs per machine, in which case this "spreading" of CPU cycles occurs naturally as VMs compete for CPU.

In summary, this chapter has presented mechanisms to independently scale resources available to individual VMs. This ability is crucial to allow DieCast to deal with heterogeneity, both within the physical machines in a given target system, and between the physical machines in the original system and the physical machines in the test harness. Taking a look at Table 5.1, we know have mechanisms that enable us to perform all the resource scaling outlined in the final row of the table. At this point, we have all the machinery required for DieCast to work. We now focus our attention on validating DieCast itself.

Chapter 5, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2008. Gupta, Diwaker; Vishwanath, Kashi V.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 5, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2006. Gupta, Diwaker; Yocum, Kenneth; McNett, Marvin; Snoeren, Alex C.; Vahdat, Amin; Voelker, Geoffrey M. The dissertation author was the primary investigator and author of this paper.

Chapter 6

DieCast Evaluation

We outlined the following goals for DieCast in Chapter 3: fidelity, reproducibility and efficiency. The approach we have taken thus far can be summarized as follows. We first map the physical machines into virtual machines and consolidate the VMs on a smaller number of physical machines, for efficiency. Using a combination of network emulation and workload generation, we can ensure reproducibility. Finally, to restore fidelity and preserve the resource equivalence between VMs in the test system and physical machines in the original system, we use time dilation in conjunction with resource scaling.

After going through the above steps, the claim is that the test system closely resembles the original system. This chapter aims to validate this claim. We begin by describing the general methodology for validating DieCast, followed by a detailed validation using three very different distributed systems. We conclude the chapter with a case study of using DieCast on a real-world commercial file system. Since time dilation does not scale main memory capacity, a DieCast-scaled system will end up partitioning the main memory among co-located VMs. This partitioning limits the number of VMs that can be created on a single physical machine. The next chapter addresses this limitation.

6.1 Methodology

We seek to answer the following questions with respect to DieCast-scaling:

1. Can we configure a smaller number of physical machines to match the CPU capacity, complex network topology, and I/O rates of a larger service? (validates efficiency)

2. How well does the performance of a scaled service running on fewer resources match the performance of a baseline service running with more resources? (validates fidelity)
3. What are the limits of scaling? At what point do our predictions lose so much accuracy that they are no longer valuable?

With respect to validating fidelity, we are interested in not just the application-specific performance metrics, but also the low-level system behavior such as the resource utilization profile. To evaluate the accuracy of DieCast scaling, We consider three different systems, representative of a broad spectrum of network services that we wish to target:

- BitTorrent [7], a popular peer-to-peer file sharing program;
- RUBiS [37], an auction service prototyped after eBay; and
- Isaac, our configurable network three-tier service that allows us to generate a range of workload scenarios.

The methodology for validating DieCast is similar to the methodology we used for validating time dilation (Section 4.4). To evaluate DieCast for a given system, we first establish the baseline performance: this involves determining the configuration(s) of interest, fixing the workload, and benchmarking the performance. We then scale the system down by an order of magnitude and compare the DieCast performance to the baseline. While we have extensively evaluated DieCast implementations for several versions of Xen, we only present the results for the Xen 3.1 implementation here.

In general, our methodology (which we believe is a "best-practice" approach) for evaluating systems with DieCast is three pronged:

1. Establish the baseline: involves determining the configuration space of interest, determining the metrics of interest, estimating variability and fixing the workload.
2. Microbenchmark DieCast: validate DieCast across minor configuration changes, for small setups.
3. Evaluate: scale the system down by an order of magnitude and use DieCast to explore the parameter space.

Each physical machine in our testbed is a dual-core 2.3-GHz Intel Xeon with 4 GB RAM. Note that since the DiskSim integration only works with fully virtualized VMs, for a fair evaluation it is *required* that even the baseline system run on VMs—ideally the baseline would be run on physical machines directly (for the para-virtualized setup, we have evaluated DieCast with physical machines as the baseline. We refer the reader to our technical report [55] for details). We configure DiskSim to emulate a Seagate ST3217 disk drive. For the baseline, DiskSim runs as usual (no requests are scaled) and with DieCast, we scale each request as described in Section 5.3.

We configure each virtual machine with 256 MB RAM and run Debian Etch on Linux 2.6.17. Unless otherwise stated, the baseline configuration consists of 40 physical machines hosting a single VM each. We then compare the performance characteristics to runs with DieCast on four physical machines hosting 10 VMs each, scaled by a factor of 10. We use ModelNet for the network emulation, and appropriately scale the link characteristics for DieCast. For allocating CPU, we use our modified Credit CPU scheduler as described in Section 5.3.

6.2 BitTorrent

We begin by using DieCast to evaluate BitTorrent [7] — a popular P2P application. For our baseline experiments, we run BitTorrent (version 3.4.2) on a total of 40 physical machines, each hosting a single virtual machine. We configure the machines to communicate across a ModelNet-emulated dumbbell topology (Figure 6.1), with varying bandwidth and latency values for the access link (A) from each client to the dumbbell and the dumbbell link itself (C). We vary the total number of clients, the file size, the network topology, and the version of the BitTorrent software. We use the distribution of file download times across all clients as the metric for comparing performance. The aim here is to observe how closely DieCast-scaled experiments reproduce behavior of the baseline case for a variety of scenarios.

The first experiment establishes the baseline where we compare different configurations of BitTorrent sharing a file across a 10-Mbps dumbbell link and constrained access links of 10 Mbps. All links have a one-way latency of 5 ms. We run a total of 40 clients (with half on each side of the dumbbell). Figure 6.2 plots the cumulative distribution of transfer times across all clients for different file sizes (10 MB and 50 MB). We show the base case using solid lines and use dashed lines to represent the DieCast-scaled case. With DieCast scaling, the distribution of download times closely matches the behavior of the original system. For instance, well-connected clients on the same side of the dumbbell as the randomly chosen seeder finish more quickly than the clients that must compete for scarce resources across the

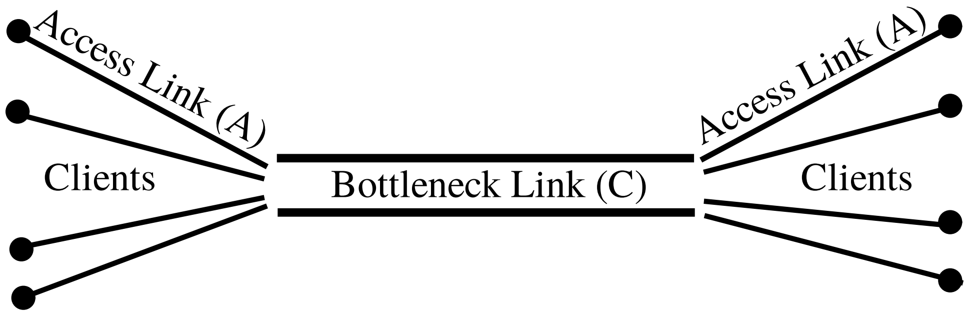


Figure 6.1: Topology for BitTorrent experiments

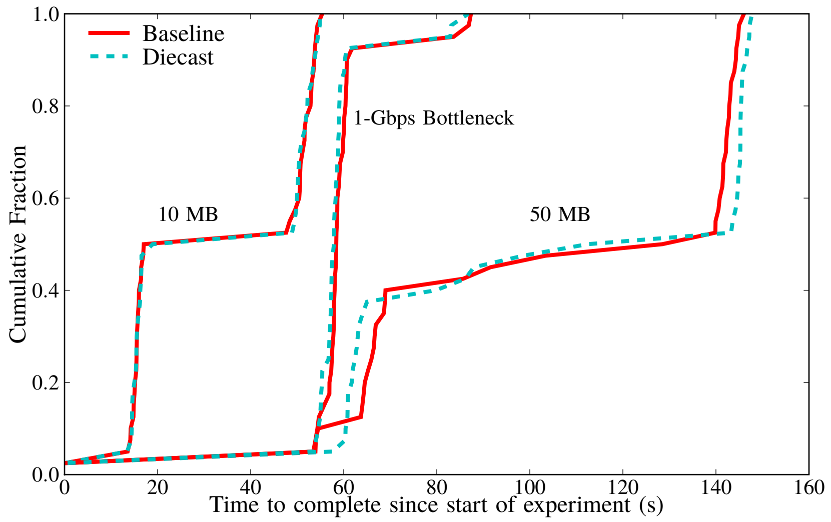


Figure 6.2: Performance with varying file sizes.

dumbbell.

Having established a reasonable baseline, we next consider sensitivity to changing system configurations. We first vary the network topology by leaving the dumbbell link unconstrained (1 Gbps) with results in Figure 6.2. The graph shows the effect of removing the bottleneck on the finish times compared to the constrained dumbbell-link case for the 50-MB file: all clients finish within a small time difference of each other as shown by the middle pair of curves. Next, we consider the behavior of a newer BitTorrent implementation (version 4.4.0) and retain the same topology from the baseline experiment (i.e., bottleneck link of 10-Mbps capacity). Figure 6.3 also shows these results. It shows two pairs of curves, with each pair corresponding to one of protocols in both the scaled and unscaled case. Once

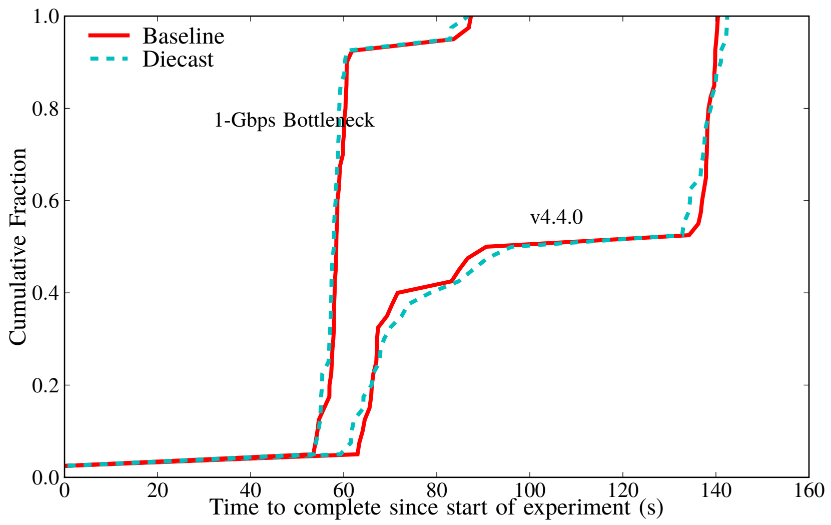


Figure 6.3: Varying topology and version.

again, the DieCast-scaled version of the experiment performs nearly identically to the baseline configuration.

Next, we consider the effect of varying the total number of clients. Using the topology from the baseline experiment we repeat the experiments for 80 and 200 simultaneous BitTorrent clients. Figure 6.4 shows the results. The curves for the baseline and DieCast-scaled versions almost completely overlap each other for 80 clients (left pair of curves) and show minor deviation from each other for 200 clients (right pair of curves). Note that with 200 clients, the bandwidth contention increases to the point where the dumbbell bottleneck becomes less important.

So far we have only considered cases where the test harness had enough resources such that scaling the system by the number of VMs on a single physical machine was sufficient. That is, we could run 10 VMs each on the four physical machines with a TDF of 10, to match the resource capacity of the 40 physical machines in the original system. However, if the machines in the test harness are less capable, then a scaling factor of 10 might not suffice. Similarly, if the machines in the test harness are significantly more powerful, then fewer machines could be used to support the 40 VMs.

This next experiment demonstrates the flexibility of DieCast to reproduce system performance under a variety of resource configurations starting with the same baseline. Figure 6.5 shows that in addition to matching 1:10 scaling using 4 physical machines hosting 10 VMs

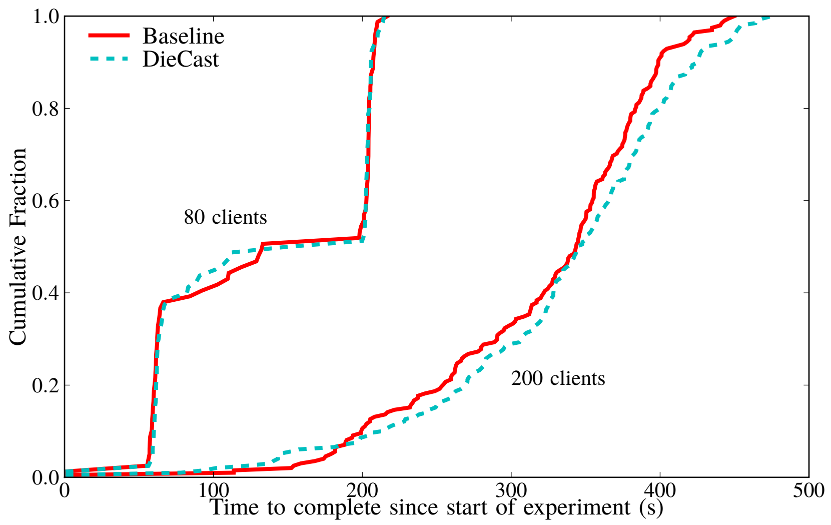


Figure 6.4: Varying number of clients.

each, we can also match an alternate configuration of 8 physical machines, hosting five VMs each with a dilation factor of five. This experiment demonstrates that even if it is necessary to vary the number of physical machines available for testing, it may still be possible to find an appropriate scaling factor to match performance characteristics. This graph also has a fourth curve, labeled "No DieCast", corresponding to running the experiment with 40 VMs on four physical machines, each with a dilation factor of 1—disk and network are *not* scaled (thus match the baseline configuration), and all VMs are allocated equal shares of the CPU. This corresponds to the approach of simply multiplexing a number of virtual machines on physical machines without using DieCast. The graph shows that the behavior of the system under such a naïve approach varies widely from actual behavior.

The above experiment clearly demonstrates that regular multiplexing fails to provide the resource equivalence required for testing distributed systems. However, it is not clear if all of the mechanisms in DieCast are necessary. In particular, it is tempting to establish the minimal possible set of mechanisms that would be able to accurately recreate a given target system. For instance, is it the case that the disk I/O scaling is always required, or can we eliminate it (making the system much simpler) without sacrificing accuracy?

Our final experiment establishes the need for disk scaling in systems where disk I/O dominates. For the baseline, we configure 10 physical machines hosting a single VM each (TDF 1) to download a 50-MB file using BitTorrent. Each machine has a direct connection

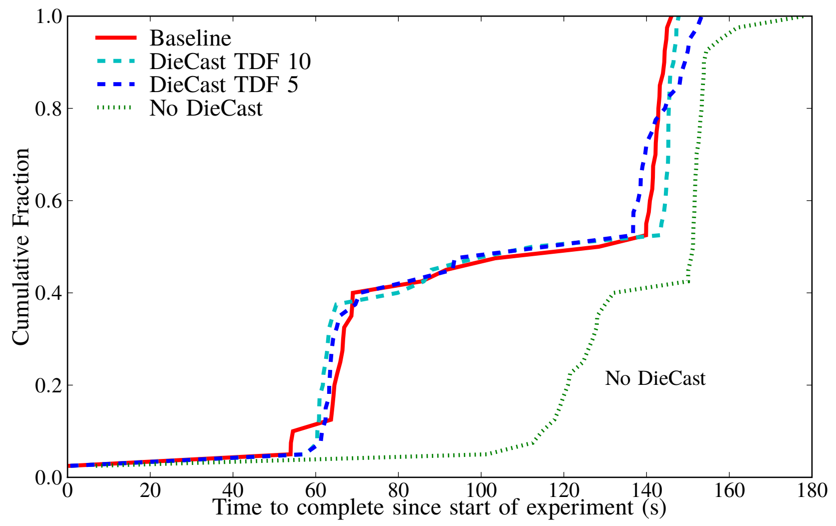


Figure 6.5: Different configurations.

to every other physical machine, that is, they create a mesh network. We do this to minimize any network routing overheads. Each link is 100 Mbps and one way latency of 1 ms. We measure the time taken to download the file averaged across all clients as well as the standard deviation. In the wide-area setting, the one way latency increases to 100 ms. We next repeat the experiment (both LAN and WAN settings) under a dilation factor of 10: all 10 VMs are running on a single physical machine, with CPU and network resources scaled appropriately. However, we do *not* scale disk I/O requests. Finally, we repeat the experiment with all resources (including disk I/O) scaled appropriately. For each experiment, we report the mean and standard deviation across the download times of the clients.

Table 6.1 shows that when disk I/O is not scaled in the LAN setting, the clients experience a 20% deviation from the baseline. The same experiment, when performed in a wide-area setting, reveals that the importance of disk scaling diminishes as network latencies begin to dominate. However, note that despite the lack of disk I/O scaling, the deviations from baseline are modest (compared to the pure I/O workload from Figure 5.14). As the experiment above demonstrates, systems with wide-area latencies are typically unaffected by disk scaling since network latencies are the dominant bottlenecks.

With DieCast, we have taken an approach of broad applicability over minimalism. The set of mechanisms in DieCast are able to accurately replicate a wide variety of systems. While it is possible that a subset of these mechanisms are sufficient for a given workload, we

Table 6.1: Time taken (in seconds) to download a 50 MB file averaged across ten BitTorrent clients. Disk I/O scaling matters more in systems with significant disk I/O. In particular, if network latencies dominates, the impact of disk scaling diminishes.

Configuration	**Baseline**	**Disk not scaled**	**Disk scaled**
LAN setting	47.71 (15.47)	38.67 (8.09)	47.67 (13.68)
WAN setting	74.98 (4.76)	73.03 (4.06)	71.35 (4.68)

strongly believe that the effort required to identify the precise, minimal set of mechanisms that is adequate in a specific case is not justified. In all subsequent experiments, all of the resource scaling mechanisms, including disk I/O scaling, are active.

6.3 RUBiS

Next, we investigate DieCast's ability to scale a fully functional Internet service. We use RUBiS [37]—an auction site prototype designed to evaluate scalability and application server performance. We chose RUBiS because of the following reasons:

- its source code is freely available;
- it ships with a highly configurable workload generator;
- it offers sufficient complexity in terms of distributed component interaction to demonstrate our ability to scale complex network services; and
- it has been used by other researchers to approximate realistic Internet Services [42, 37, 32].

We use the PHP implementation of RUBiS running Apache as the web server and MySQL as the database. For consistent results, we re-create the database and pre-populate it with 100,000 users and items before each experiment. We use the default read-write transaction table for the workload that exercises all aspects of the system such as adding new items, placing bids, adding comments, viewing and browsing the database. The RUBiS workload generators warm up for 60 seconds, followed by a session run time of 600 seconds and ramp down for 60 seconds.

We emulate a topology of 40 nodes consisting of 8 database servers, 16 web servers and 16 workload generators as shown in Figure 6.6. A 100-Mbps network link connects

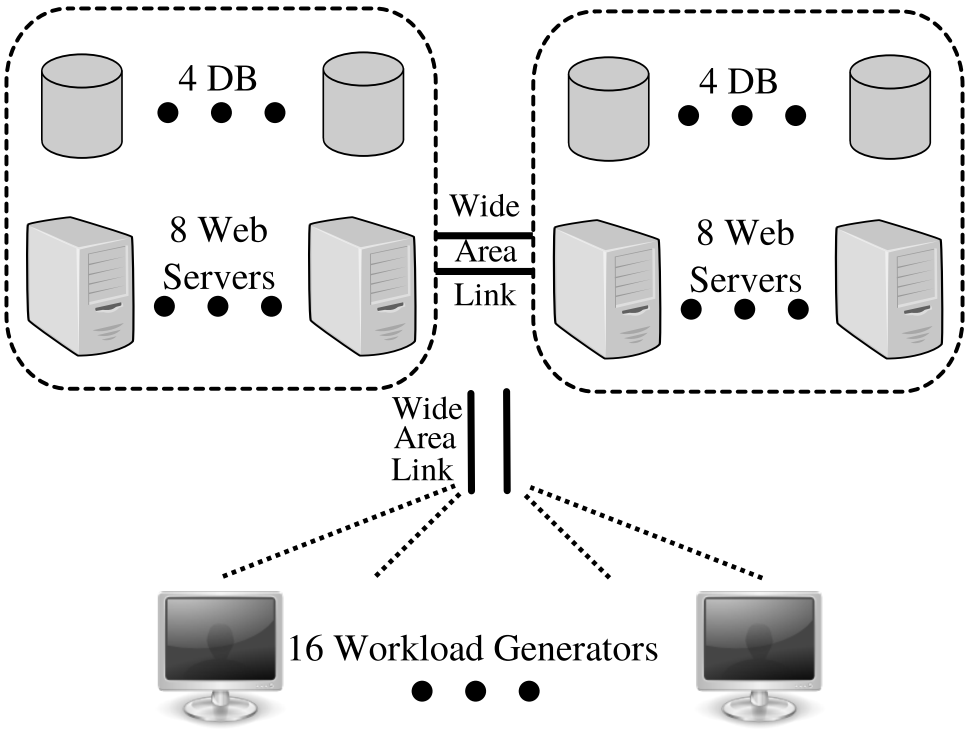
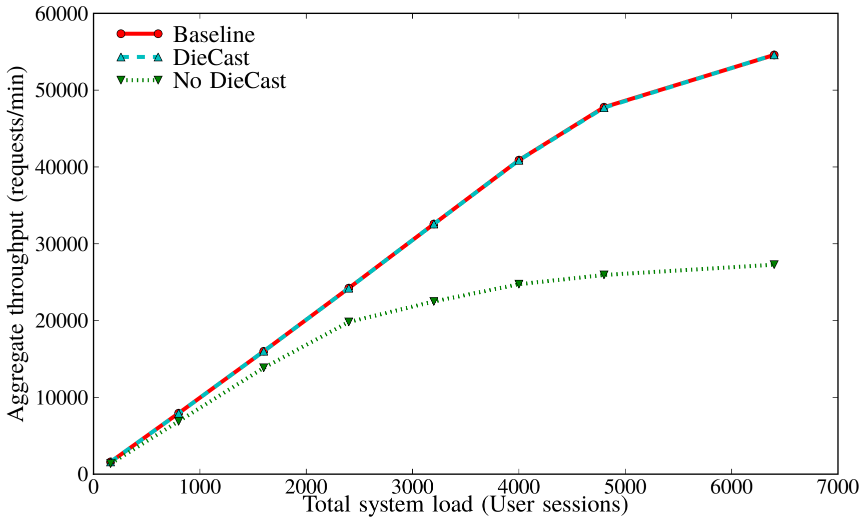


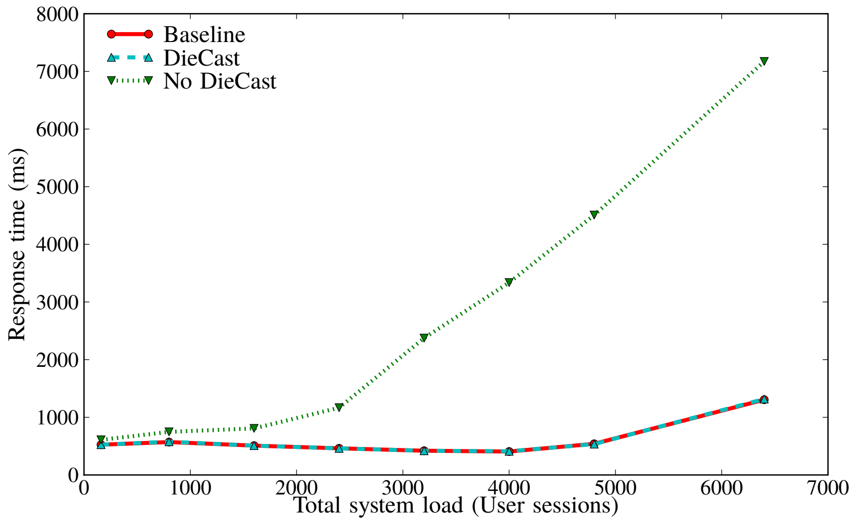
Figure 6.6: RUBiS setup.

two replicas of the service spread across the wide-area at two sites. Within a site, 1-Gbps links connect all components. For reliability, half of the web servers at each site use the database servers in the other site. The communication patterns between the web servers and the database servers have been setup to model real life scenarios (avoid correlated failures, network partitioning etc.) as well as exercise the network. There is one load generator per web server and all load generators share a 100-Mbps access link. Each system component (servers, workload generators) runs in its own Xen VM.

We now evaluate DieCast's ability to predict the behavior of this RUBiS configuration using fewer resources. Figures 6.7a and 6.7b compare the baseline performance with the scaled system for overall system throughput and average response time (across all client-webserver combinations) on the Y-axis as a function of number of simultaneous clients (offered load) on the X-axis. In both cases, the performance of the scaled service closely tracks that of the baseline. We also show the performance for the "No DieCast" configuration: regular VM multiplexing with no DieCast-scaling. Without DieCast to offset the resource contention, the aggregate throughput drops with a substantial increase in response times. Interestingly, for one of our initial tests, we ran with an unintended mis-configuration of the RUBiS database:



(a) Throughput



(b) Response time

Figure 6.7: Comparing RUBiS application performance: Baseline vs. DieCast.

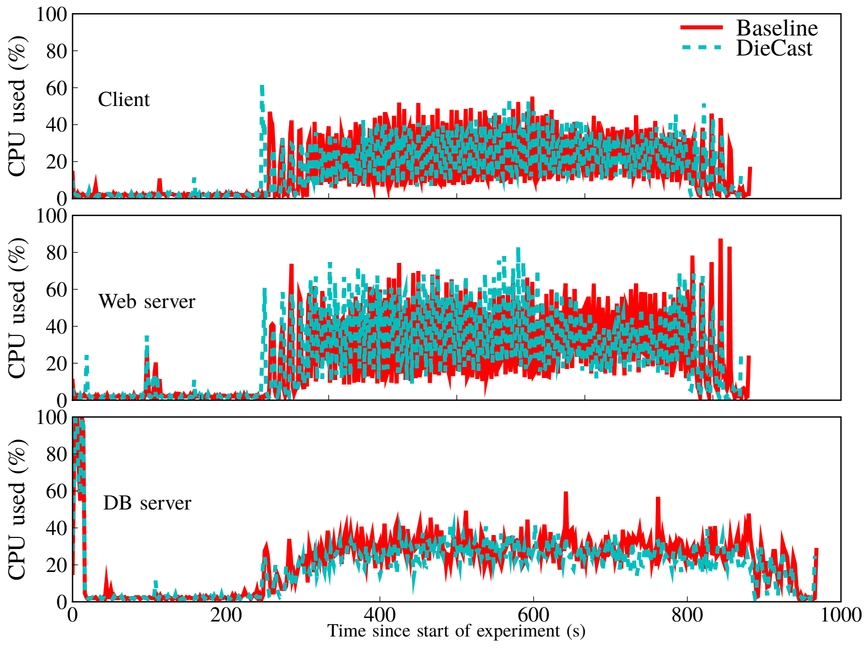


Figure 6.8: CPU profile.

the workload had commenting-related operations enabled, but the relevant tables were missing from the database. This setup led to an approximately 25% error rate with similar timings in the responses to clients in both the baseline and DieCast configurations. Thus, for this experiment DieCast did not mask misconfigurations that were present in the original system. These types of configuration errors are one example of the types of testing that we wish to enable with DieCast.

Next, Figures 6.8 and 6.9 compare CPU and memory utilizations for both the scaled and unscaled experiments as a function of time for the case of 4800 simultaneous user sessions: we pick one node of each type (database server, web server, load generator) at random from the baseline, and use the same three nodes for comparison with DieCast. One important question is whether the average performance results in earlier figures hide significant incongruities in per-request performance. Here, we see that resource utilization in the DieCast-scaled experiments closely tracks the utilization in the baseline on a per-node and per-tier (client, web server, database) basis. Similarly, Figure 6.10 compares the network utilization of individual links in the topology for the baseline and DieCast-scaled experiment. We sort the links by the amount of data transferred per link in the baseline case. This graph demonstrates that DieCast closely tracks and reproduces variability in network utilization for various hops in the topology. For

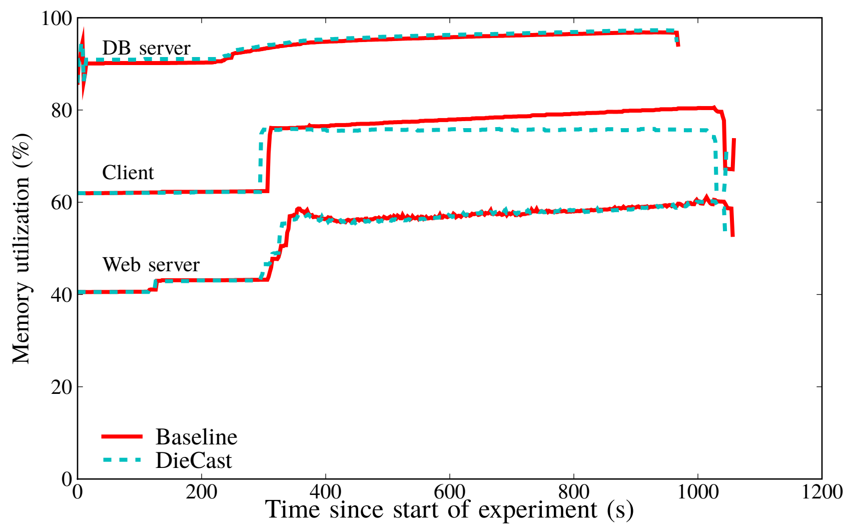


Figure 6.9: Memory profile.

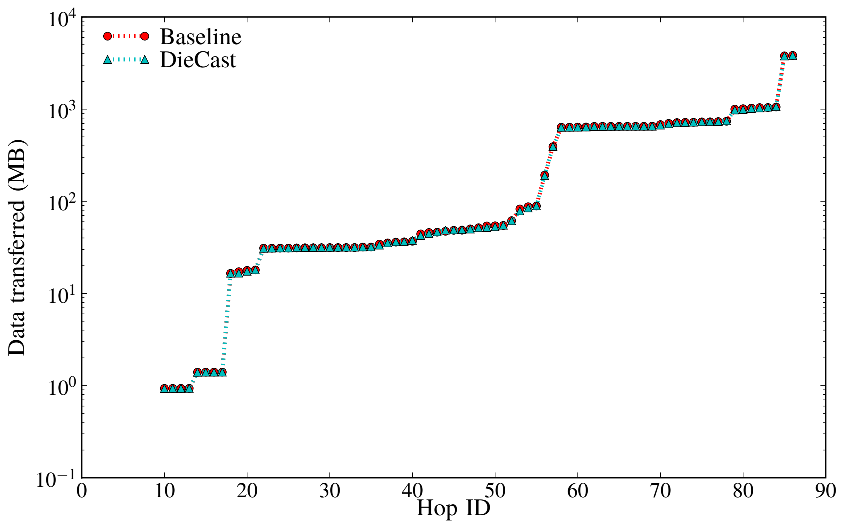


Figure 6.10: Network profile.

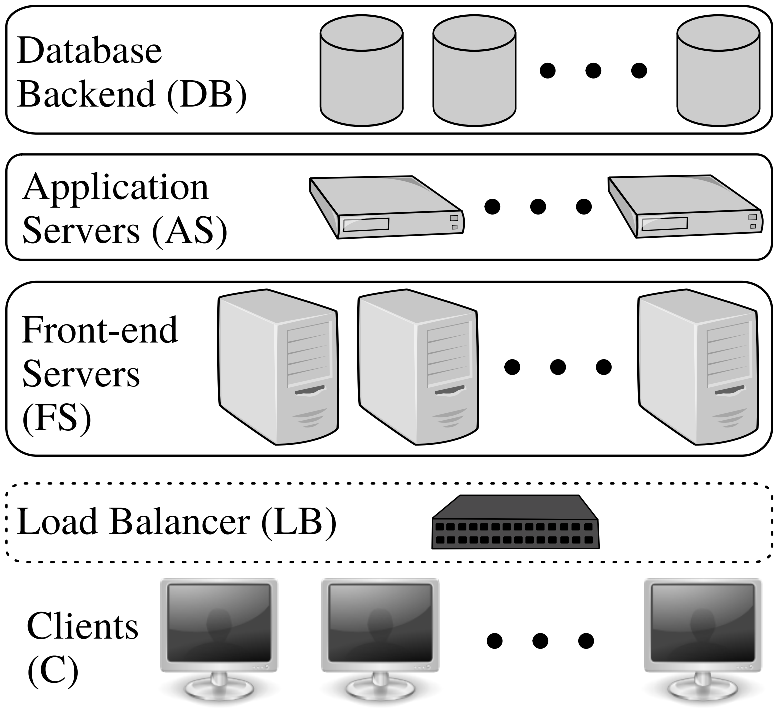


Figure 6.11: Architecture of Isaac.

instance, hops 86 and 87 in the figure correspond to access links of clients and show the maximum utilization, whereas individual access links of Webservers are moderately loaded.

6.4 Exploring DieCast Accuracy

While we were encouraged by DieCast's ability to scale RUBiS and BitTorrent, they represent only a few points in the large space of possible network service configurations, for instance, in terms of the ratios of computation to network communication to disk I/O. Hence, we built Isaac, a configurable multi-tier network service to stress the DieCast methodology on a range of possible configurations. Figure 6.11 shows Isaac's architecture. Requests originating from a client (C) travel to a unique front-end server (FS) via a load balancer (LB). The FS makes a number of calls to other services through application servers (AS). These application servers in turn may issue read and write calls to a database back end (DB) before building a response and transmitting it back to the front end server, which finally responds to the client. Both the AS and the FS might do arbitrarily long computations (implemented by computing SHA-1 hashes repeatedly) on the results obtained from the individual services they

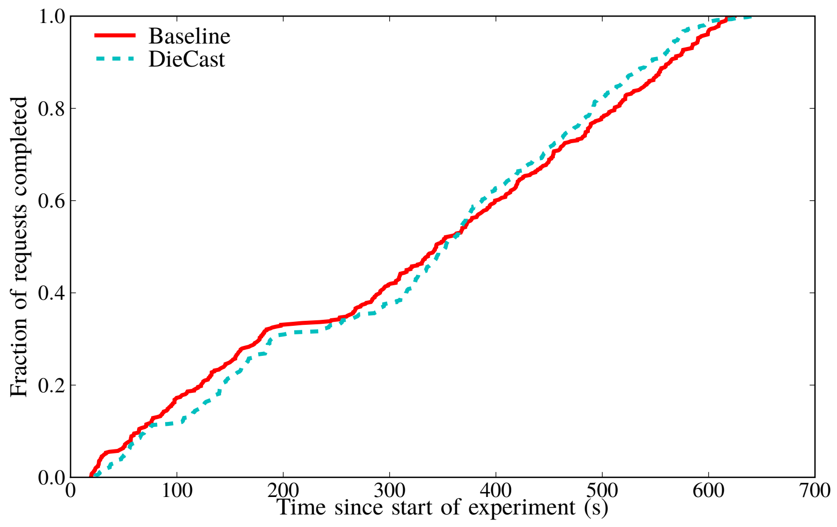


Figure 6.12: Request completion time.

communicate with before returning a response to the client.

Isaac is written in Python and allows configuring the service to a given interconnect topology, computation, communication, and I/O pattern. A configuration describes, on a per-request class basis, the computation, communication, and I/O characteristics across multiple service tiers. In this manner, we can configure experiments to stress different aspects of a service and to independently push the system to capacity along multiple dimensions. We use MySQL for the database tier to reflect a realistic transactional storage tier. It is worth noting here that the aim of Isaac is not to capture a realistic or representative Internet service. Instead, Isaac aims to capture the architectural complexity of multi-tier Internet services and, at the same time, lets us reason about the limits of DieCast when subject to extreme conditions for one particular configuration of an Internet service.

For our first experiment, we configure Isaac with four DBs, four ASs, four FSs and 28 clients. The clients generate requests, wait for responses, and sleep for some time before generating new requests. Each client generates 20 requests and each such request touches five ASs (randomly selected at run time) after going through the FS. Each request from the AS involves 10 reads from and 2 writes to a database each of size 1KB. The database server is also chosen randomly at runtime. Upon completing its database queries, each AS computes 500 SHA-1 hashes of the response before sending it back to the FS. Each FS then collects responses from all five AS's and finally computes 5,000 SHA-1 hashes on the concatenated results before

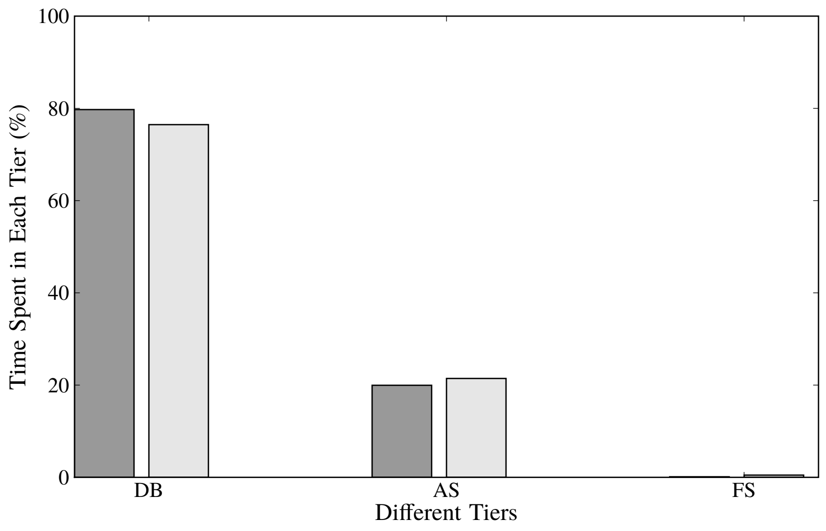


Figure 6.13: Tier-breakdown.

replying to the client. In later experiments, we vary both the amount of computation and I/O to quantify sensitivity to varying resource bottlenecks

We perform this 40-node experiment both with and without DieCast. In all cases, performance of the DieCast-scaled system matched closely with the baseline case. We omit these results in favor of presenting some more interesting scenarios. Rather, we run a more complex experiment where a subset of the machines fail and then recover. Our goal is to show that DieCast can accurately match application performance before the failure occurs, during the failure scenario, and the application's recovery behavior. After 200 seconds, we fail half of the database servers (chosen at random) by stopping MySQL servers on the corresponding nodes. As a result, client requests accessing failed databases will not complete, slowing the rate of completed requests. After one minute of downtime, we restart the MySQL server and soon after we expect to see the request completion rate to regain its original value.

Figure 6.12 shows fraction of requests completed on the Y-axis as a function of time since the start of the experiment on the X-axis. DieCast closely matches the baseline application behavior with a dilation factor of 10. We also compare the percentage of time spent in each of the three tiers of Isaac averaged across all requests. Figure 6.13 shows that in addition to the end-to-end response time, DieCast closely tracks the system behavior on a per-tier basis. For instance, we see that the database backend is the bottleneck in this experiment as each request spends approximately 80% of its time in this TIER.

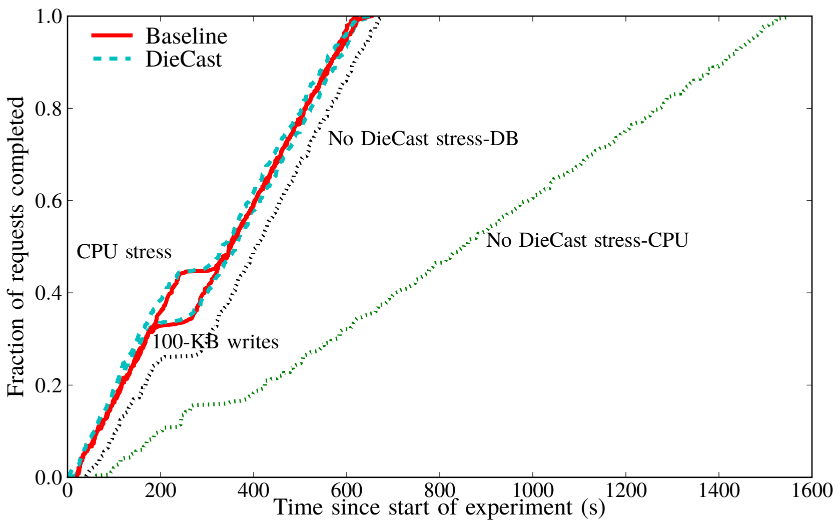


Figure 6.14: Stressing database and CPU.

Encouraged by the results of the previous experiment, we next attempt to saturate individual components of Isaac to explore the limits of DieCast's accuracy. In other words, we want to examine if the DieCast approach breaks down if the system under test is subject to high levels of resource utilization. First, we evaluate DieCast's ability to scale network services when database access dominates per-request service time. Figure 6.14 shows the completion time for requests, where each service issues a 100-KB (rather than 1-KB) write to the database with all other parameters remaining the same. The larger requests translate to a total of 1 MB of database writes for every request from a client. resulting in roughly 140 KBps of disk I/O at each of the database servers (the maximum throughput for the simulated disk is approximately 750 KBps). Even for these larger data volumes, DieCast faithfully reproduces system performance. While for this workload, we are able to maintain good accuracy, the evaluation of disk dilation summarized in Figure 5.14 in the previous chapter suggests that there will certainly be points where disk dilation inaccuracy will affect overall DieCast accuracy.

Next, we evaluate DieCast accuracy when one of the components in our architecture saturates the CPU. Specifically, we configure our front-end servers such that prior to sending each response to the client, they compute SHA-1 hashes of the response 500,000 times to artificially saturate the CPU of this tier. The results of this experiment are also shown in Figure 6.14. We are encouraged overall as the system does not significantly diverge even to the point of CPU saturation. For instance, the CPU utilization for nodes hosting the FS in this experiment varied from $50 - 80\%$ for the duration of the experiment and even under such conditions DieCast closely matched the baseline system performance. The "No DieCast" lines plot the performance of the configurations where we stress the database and the CPU independently, under regular VM multiplexing (without DieCast-scaling). As with BitTorrent and RUBiS, we see that without DieCast, the test infrastructure fails to predict the performance of the baseline system.

This completes our evaluation of the DieCast approach across three widely varying workloads in a range of scenarios. Encouraged by these results we present our experience with deploying DieCast in a real-world setting in the next section.

6.5 Commercial System Evaluation

While we are encouraged by DieCast's accuracy for the applications we considered in the previous sections, all of the experiments were designed by us and were largely academic in nature. To understand the generality of our system, we consider its applicability to a large-scale commercial system.

Panasas [16] builds scalable storage systems targeting Linux cluster computing environments. It supplies solutions to numerous government agencies and research organizations, oil and gas companies, media companies and several commercial HPC enterprises. A core component of Panasas's products is the PanFS parallel filesystem (henceforth referred to as PanFS): an object-based cluster filesystem that presents a single, cache coherent unified namespace to clients.

To meet customer requirements, Panasas must ensure its systems can deliver appropriate performance under a range of client access patterns. Unfortunately, it is often impossible to create a test environment that reflects the setup at a customer site. Since Panasas has several customers with very large super-computing clusters and limited test infrastructure at its disposal, its ability to perform testing at scale is severely restricted by hardware availability; exactly the type of situation DieCast targets. For example, the Los Alamos National Lab has deployed PanFS with its Roadrunner peta-scale super computer [17]. The Roadrunner system is designed to deliver a sustained performance level of one petaflop at an estimated cost of $90 million. Because of the tremendous scale and cost, Panasas cannot replicate this computing environment for testing purposes.

Porting Time Dilation. In evaluating our ability to apply DieCast to PanFS, we encountered one primary limitation. PanFS clients use a Linux kernel module to communicate with the PanFS server. The client-side code runs on recent versions of Xen , and hence DieCast supported them with no modifications. However, the PanFS server runs in a custom operating system derived from an older version of FreeBSD that does not support Xen. The significant modifications to the base FreeBSD operating system made it impossible to port PanFS to a more recent version of FreeBSD that does support Xen. Ideally, it would be possible to simply encapsulate the PanFS server in a fully virtualized Xen VM. However, recall that this requires virtualization support in the processor which was unavailable in the hardware Panasas was using. Even if we had the hardware, Xen did not support FreeBSD on fully-virtualized VMs until recently due to a well-known bug [10]. Thus, unfortunately we could not easily employ the existing time dilation techniques with PanFS on the server side. However, since we believe DieCast concepts are general and not restricted to Xen, we took this opportunity to explore whether we could modify the PanFS OS to support DieCast, without any virtualization support.

Implementing time dilation requires, among other things, slowing down the passage of time, slowing down the rate of interrupts, and uniformly scaling any external time sources that could "leak" information about the real time to the OS. To implement time dilation in the PanFS kernel, we scale the various time sources (such as the PIT and the TSC register), and consequently, the wall clock. The TDF can be specified at boot time as a kernel parameter. As before, we need to scale down resources available to PanFS such that its perceived capacity matches the baseline.

Recall that time dilation uniformly scales the amount of resources perceived to be available to the OS. Hence, when running under dilation, PanFS would perceive a uniformly faster network, disk and CPU. However, our goal is to evaluate the performance of a non-dilated PanFS system against a large number of clients. We therefore need to ensure that the dilated PanFS server perceives similar resource characteristics as the non-dilated server. This requires mechanisms analogous to those described in Chapter 5. Note that it is much more critical to preserve the disk I/O characteristics in a high performance storage system than in other applications. Without any kind of disk scaling, the dilated file servers would perceive a much faster local disk, thus affecting the client perceived latency.

For disk dilation, we were faced by the complication that multiple hardware and software components interact in PanFS to service clients. For performance, there are several parallel data paths and many operations are either asynchronous or cached. Accurately implementing disk dilation would require accounting for all of the possible code paths as well as modeling the disk drives with high fidelity. In an ideal implementation, if the physical service time for a disk request is s and the TDF is t, then the request should be delayed by time $(t-1)s$ such that the total physical service time becomes $t \times s$, which under dilation would be perceived as the desired value of s.

Unfortunately, the Panasas operating system only provides coarse-grained kernel timers. Consequently, sleep calls with small durations tend to be inaccurate. Using a number of micro-benchmarks, we determined that the smallest sleep interval that could be accurately implemented in the PanFS operating system was 1 ms.

This limitation affects the way disk dilation can be implemented. For I/O intensive workloads, the rate of disk requests is high. At the same time, the service time of each request is relatively modest. In this case, delaying each request individually is not an option, since the overhead of invoking sleep dominates the injected delay and gives unexpectedly large slowdowns. Thus, we chose to aggregate delays across some number of requests whose service time sums to more than 1 ms and periodically inject delays rather than injecting a delay for each request. Another practical limitation is that it is often difficult to accurately bound the service time of a disk request. This is a result of the various I/O paths that exist: requests can be synchronous or asynchronous, they can be serviced from the cache or not and so on.

While we realize that this implementation is imperfect, it works well in practice and can be automatically tuned for each workload. A perfect implementation would have to accurately model the low-level disk behavior and improve the accuracy of the kernel sleep function. Because operating systems and hardware will increasingly support native virtualization, we feel that our simple disk dilation implementation targeting individual PanFS workloads is reasonable in practice to validate our approach.

For scaling the network, we use Dummynet [89], which ships as part of the PanFS OS. However, there was no mechanism for limiting the CPU available to the OS, or to slow the disk. The PanFS OS does not support non work-conserving CPU allocation. Further, simply modifying the CPU scheduler for user processes is insufficient because it would not throttle the rate of kernel processing. For CPU dilation, we had to modify the kernel as follows. We created a CPU-bound task, (`idle`), in the kernel and we statically assigned it the highest scheduling priority. We scale the CPU by maintaining the required ratio between the run times of the `idle` task and all remaining tasks. If the `idle` task consumes sufficient CPU, it is

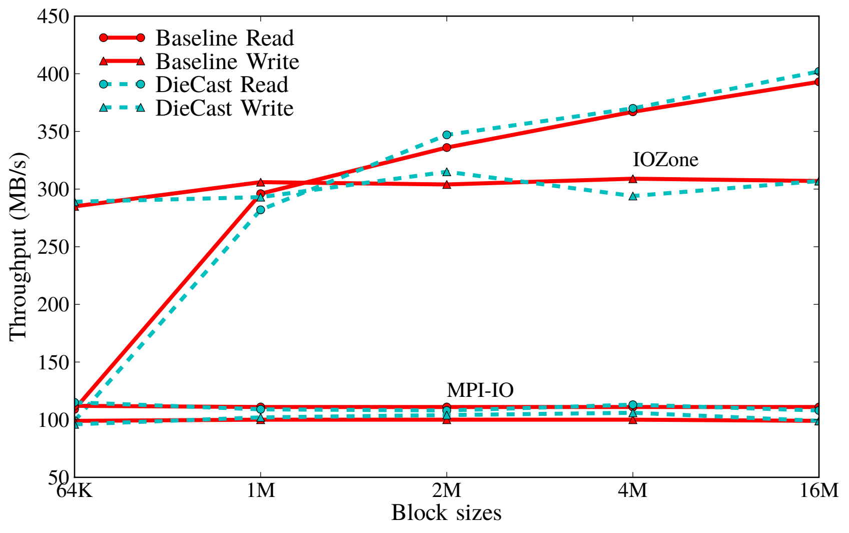


Figure 6.15: Validating DieCast on PanFS.

removed from the run queue and the regular CPU scheduler kicks in. If not, the scheduler always picks the `idle` task because of its priority.

Validation We first wish to establish DieCast accuracy by running experiments on bare hardware and comparing them against DieCast-scaled virtual machines. We start by setting up a storage system consisting of a PanFS server with 20 disks of capacity 250 GB each (5 TB total storage). We evaluate two benchmarks from the standard bandwidth test suite used by Panasas. The first benchmark involves 10 clients (each on a separate machine) running IOZone [11]. The second benchmark uses the Message Passing Interface (MPI) across 100 clients (again, on separate machines) [54].

For DieCast scaling, we repeat the experiment with our modifications to the PanFS server configured to enforce a dilation factor of 10. Thus, we allocate 10% of the CPU to the server and dilate the network using Dummynet to 10% of the physical bandwidth and 10 times the latency (to preserve the bandwidth-delay product). On the client side, we have all clients running in separate virtual machines (10 VMs per physical machine), each receiving 10% of the CPU with a dilation factor of 10.

Figure 6.15 plots the aggregate client throughput for both experiments on the Y-axis as a function of the data block size on the X-axis. Circles mark the read throughput while triangles mark write throughput. We use solid lines for the baseline and dashed lines for the

Table 6.2: Aggregate read/write throughputs from the IOZone benchmark with block size 16 MB. PanFS performance scales gracefully with larger client populations.

Aggregate Throughput	**Number of clients**		
	10	250	1000
Write	370 MB/s	403 MB/s	398 MB/s
Read	402 MB/s	483 MB/s	424 MB/s

DieCast-scaled configuration. For both reads and writes, DieCast closely follows baseline performance, never diverging by more than 5% even for unusually large block sizes.

Scaling With sufficient faith in the ability of DieCast to reproduce performance for real-world application workloads we next aim to push the scale of the experiment beyond what Panasas can easily achieve with their existing infrastructure.

We are interested in the scalability of PanFS as we increase the number of clients by two orders of magnitude. To achieve this, we design an experiment similar to the one above, but this time we fix the block size at 16 MB and vary the number of clients. We use 10 VMs each on 25 physical machines to support 250 clients to run the IOZone benchmark. We further scale the experiment by using 10 VMs each on 100 physical machines to go up to 1000 clients. In each case, all VMs are running at a TDF of 10. The PanFS server also runs at a TDF of 10 and all resources (CPU, network, disk) are scaled appropriately. Table 6.2 shows that the performance of PanFS with increasing client population. Interestingly, we find relatively little increase in throughput as we increase the client population. Upon investigating further, we found that a single PanFS server configuration is limited to 4 Gb/s (500 MB/s) of aggregate bisection bandwidth between the servers and clients (including any IP and filesystem overhead). While our network emulation accurately reflected this bottleneck, we did not catch the bottleneck until we ran our experiments.

We would like to emphasize that prior to our experiment, Panasas had been unable to perform experiments at this scale. This limitation is in part due to the fact that such a large number of machines might not be available at any given time for a single experiment. Further, even if machines are available, blocking a large number of machines results in significant resource contention because several other smaller experiments are then blocked on availability of resources. Our experiments demonstrate that DieCast can leverage existing resources to work around these types of problems.

6.6 DieCast Usage Scenarios

DieCast is not a panacea for testing large-scale networked services. In this section, we discuss DieCast's applicability and limitations for testing large-scale network services in a variety of environments.

DieCast aims to reproduce the performance of an original system configuration and is well suited for predicting the behavior of the system under a variety of workloads. Further, because the test system can be subject to a variety of realistic and projected client access patterns, DieCast may be employed to verify that the system can maintain the terms of service level agreements (SLA).

DieCast runs in a controlled and partially emulated network environment. Thus, it is relatively straightforward to consider the effects of revamping a service's network topology (e.g., to evaluate whether an upgrade can alleviate a communication bottleneck). DieCast can also systematically subject the system to failure scenarios. For example, system architects may develop a suite of fault-loads to determine how well a service maintains response times, data quality, or recovery time metrics. Similarly, because DieCast controls workload generation it is appropriate for considering a variety of attack conditions. For instance, it can be used to subject an Internet service to large-scale Denial-of-Service (DoS) attacks. DieCast may enable evaluation of various DOS mitigation strategies or software architectures.

Many difficult-to-isolate bugs result from system configuration errors (e.g., at the OS, network, or application level) or inconsistencies that arise from "live upgrades" of a service. The resulting faults may only manifest as errors in a small fraction of requests and even then after a specific sequence of operations. Operator errors and mis-configurations [83, 85] are also known to account for a significant fraction of service failures. We attempt to run the exact software configuration of the original system and may even run a network service while undergoing its own upgrade process.DieCast makes it possible to capture the effects of mis-configurations and upgrades before a service goes live.

At the same time, DieCast will not be appropriate for certain service configurations. As discussed earlier, DieCast is unable to scale down the memory or storage capacity of a service. Services that rely on multi-petabyte data sets or saturate the physical memories of all of their machines with little to no cross-machine memory/storage redundancy may not be suitable for DieCast testing. If system behavior depends heavily on the behavior of the processor cache, and if multiplexing multiple VMs onto a single physical machine results in significant cache pollution, then DieCast may under-predict the performance of certain application configurations. One possible exception is if the service itself can be modified to utilize less data. For instance, for certain services it may be possible to perform the same amount of I/O and computation over a smaller amount of data (on disk or in memory). In this case, the exact data responses may not be comparable to the original service but the performance and failure responses may be.

DieCast may change the fine-grained timing of individual events in the test system. Hence, DieCast may not be able to reproduce certain race conditions or timing errors in the original service. Some bugs, such as memory leaks, will only manifest after running for a significant period of time. Given that we inflate the amount of time required to carry out a test, it may take too long to isolate these types of errors using DieCast.

Multiplexing multiple virtual machines onto a single physical machine, running with an emulated network, and dilating time will introduce some error into the projected behavior of target services. This error has been small for the network services and scenarios we evaluate in this dissertation. In general however, DieCast's accuracy will be service and deployment-specific. We have not yet established an overall limit to DieCast's scaling ability. In separate experiments not reported in this dissertation, we have successfully run with scaling factors of 100. However, in these cases, the limitation of time itself becomes significant. Waiting 10 times longer for an experiment to configure is often reasonable, but waiting 100 times longer may become difficult.

Some services employ a variety of custom hardware, such as load balancing switches, firewalls, and storage appliances. In general, it may not be possible to scale such hardware in our test environment. Depending on the architecture of the hardware, one approach is to wrap the various operating systems for such cases in scaled virtual machines. Another approach is to run the hardware itself and to build custom wrappers to intercept requests and responses, scaling them appropriately. A final option is to run such hardware unscaled in the test environment, introducing some error in system performance. Our work with PanFS shows that it is feasible to scale unmodified services into the DieCast environment with relatively little work on the part of the developer.

Finally, we would like to emphasize that there is nothing special or fundamental about 1/10 scaling. DieCast can just as well be used for other scaling factors. We focus on 1/10 scaling for this work merely to demonstrate the feasibility of the approach.

Applications with coarse grained timing requirements may tolerate higher time dilation factors, while time sensitive applications might experience performance degradation. In general, components of the system that can not be perfectly scaled by DieCast—such as low level caching effects or memory access latencies—might impact DieCast accuracy at higher scaling factors.

We stated earlier that this dissertation explores alternative mechanisms for resource multiplexing in virtual machines and the applications they enable. The past few chapters presented one such mechanism — time dilation — and the innovative applications it enables, particularly in scalable network emulation and large-scale system testing. But time dilation only attacks the issue of vertical scalability. For a framework like DieCast, the ability to support more and more virtual machines on a single physical machine is extremely important. Addressing horizontal scalability is thus important as an important problem in its own right, and also important for supporting other applications like DieCast or regular server consolidation.

Another important issue that we have deliberately avoided thus far is the issue of scaling memory. Recall that time dilation only scales up resources that have a time-based component. Static resources such as main memory capacity or hard-drive capacity are not scaled. Thus, co-located VMs in a DieCast-scaled system are still limited in the amount of memory they have. This severely limits the applicability of DieCast— either the test harness should be retrofitted with additional memory, or the application should not hit a memory bottleneck while running in the memory-constrained VMs.

The above problems are related: if we can scale main memory capacity, we can support more VMs per physical machines, thus increasing horizontal scalability. We address this issue in the following chapters.

Chapter 6, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2008. Gupta, Diwaker; Vishwanath, Kashi V.; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

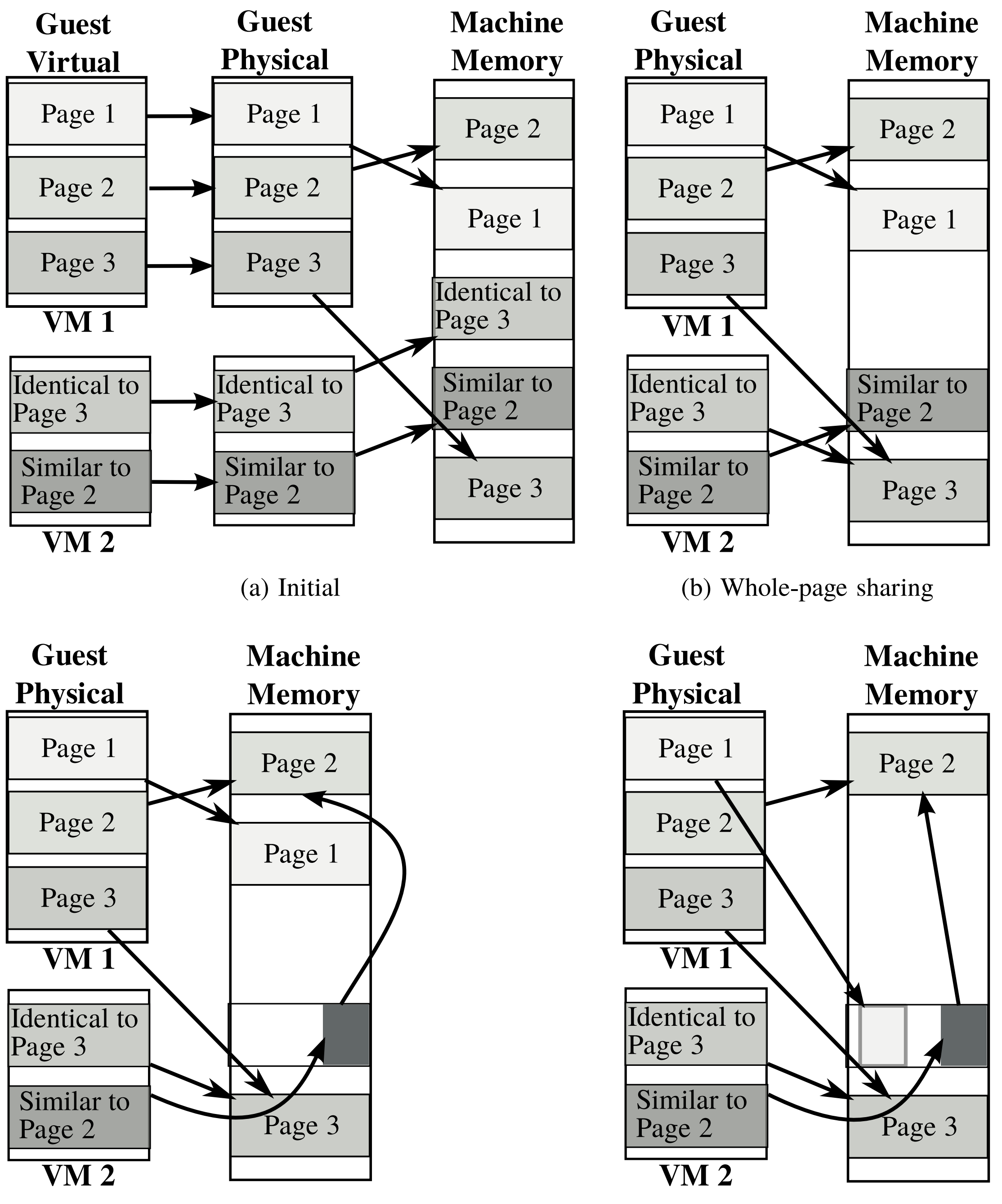
Chapter 7

Harnessing Memory Redundancy Across Virtual Machines

Section 1.3.2 described how main memory was the primary bottleneck in increasing the horizontal scalability of VM multiplexing. In this chapter, we present the design and implementation of Difference Engine, our system to more efficiently manage system memory across virtual machines. Difference Engine can support a given number of VMs using far fewer memory than that would otherwise be required, thereby freeing memory to create additional VMs.

7.1 Architecture

The key insight behind Difference Engine is that virtual machine environments present unique opportunities for memory sharing. Before describing these opportunities, let us first refresh how the VMM manages system memory. Xen and other platforms that support fully virtualized guests use a mechanism called "shadow page tables" to manage guest OS memory [110]. The guest OS has its own copy of the page table that it manages believing that they are the hardware page tables, though in reality it is just a map from the guest's virtual memory to its notion of physical memory. This map is called the virtual-to-physical map or the V2P map. In addition, Xen maintains a map from the guest's notion of physical memory to the machine memory (P2M map). The shadow page table is a cache of the results of composing the V2P map with the P2M map, mapping guest virtual memory directly to machine memory. Loosely, it is the virtualized analog to a software TLB. The shadow page table enables quick



(a) Initial

(b) Whole-page sharing

(c) Patching

(d) Compression

Figure 7.1: The three different memory conservation techniques employed by Difference Engine: whole-page sharing, page patching, and compression. In this example, five physical pages are stored in less than three machine memory pages for a savings of roughly 50%.

page translation and look-ups, and more importantly, can be used directly by the CPU.

The idea behind Difference Engine is that there might be memory pages within a VM or even across VMs that have similar content. The goal is to exploit these commonalities to represent the original set of pages using a smaller number of physical pages. Difference Engine uses three distinct mechanisms that work together to realize the benefits of memory sharing, as shown in Figure 7.1. In this example, two VMs have allocated five pages total, each initially backed by distinct pages in machine memory (Figure 7.1a). Once the pages are allocated, Difference Engine runs in the background to reduce the amount of machine memory used to store the pages. For brevity, we only show how the mapping from guest physical memory to machine memory changes; the guest virtual to guest physical mapping remains unaffected.

First, for identical pages across the VMs, we store a single copy and create references that point to the original. In Figure 7.1b, one page in VM-2 is identical to one in VM-1. For pages that are similar, but not identical, we store a patch against a reference page and discard the redundant copy. In Figure 7.1c, the second page of VM-2 is stored as a patch to the second page of VM-1. Finally, for pages that are unique and infrequently accessed, we compress them in memory to save space. In Figure 7.1d, the remaining private page in VM-1 is compressed. The actual machine memory footprint is now less than three pages, down from five pages originally.

In all three cases, Difference Engine must select candidate pages that are less likely to be accessed in the future, for efficiency. We achieve this goal using a global clock that scans memory in the background, identifying pages that have not been recently used. In addition, reference pages for sharing or patching must be found quickly without introducing performance overhead. Difference Engine uses full-page hashes and hash-based fingerprints to identify good candidates. Finally, we implement a demand paging mechanism to push VM pages to/from secondary storage in Domain-0 to support memory oversubscription.

7.1.1 Page Sharing

Difference Engine's implementation of content-based page sharing is similar to those in earlier systems [110]. We walk through memory looking for identical pages. As we scan memory, we hash each page and index it based on its hash value. Identical pages hash to the same value and a collision indicates that a potential matching page has been found. However, there is a non-zero probability that non-identical pages hash to the same value. To guard against this, we perform a byte-by-byte comparison to ensure that the pages are indeed identical before sharing them.

Upon identifying target pages for sharing, we reclaim one of the pages and update the virtual memory to point at the shared copy. Both mappings are marked read-only, so that writes to a shared page cause a page fault that will be trapped by the VMM. The VMM returns a private copy of the shared page to the faulting VM and updates the virtual memory mappings appropriately. If no VM refers to a shared page, the VMM reclaims it and returns it to the free memory pool.

7.1.2 Patching

Traditionally, the goal of page sharing has been to eliminate redundant copies of *identical* pages, considering both intra-VM and inter-VM pages. Difference Engine considers further reducing the memory required to store *similar* pages by constructing patches that represent a page as the difference relative to a reference page. To motivate this design decision, we provide an initial study into the potential savings due to sub-page sharing, both within and across virtual machines. First, we define the following two heterogeneous workloads, each involving three 512-MB virtual machines:

- Mixed-1: Windows XP SP1 hosting RUBiS [37]; Debian 3.1 compiling the Linux kernel; Slackware 10.2 compiling Vim 7.0 followed by a run of the `lmbench` benchmark [14].
- Mixed-2: Windows XP SP1 running Apache 2.2.8 hosting approximately 32,000 static web pages crawled from Wikipedia, with `httperf` running on a separate machine requesting these pages; Debian 3.1 running the SysBench database benchmark [18] using 10 threads to issue 100,000 requests; Slackware 10.2 running dbench [98] with 10 clients for six minutes followed by a run of the IOZone benchmark [11].

We designed these workloads to stress the memory saving mechanisms since opportunities for identical page sharing are reduced. Our choice of applications was guided by the VMmark benchmark [80] and the vmbench suite [78]. In this first experiment, for a variety of configurations — number of VMs, allocations, types of applications — we suspend the VMs after completing a benchmark, and consider a static snapshot of their memory to determine the number of pages required to store the images using various techniques. Table 7.1 shows the results of our analysis for the Mixed-1 workload.

Table 7.1: Effectiveness of page sharing across three 512-MB VMs running Windows XP, Debian and Slackware Linux using 4-KB pages.

Pages	**Initial**	**After Sharing**	**After Patching**
Unique	191,646	191,646	
Sharable (non-zero)	52,436	3,577	
Zero	149,038	1	
Total	393,120	195,224	88,422
Reference		50,727	50,727
Patchable		144,497	37,695

The first column breaks down these 393,120 pages into three categories: 149,038 zero pages (i.e., the page contains all zeros), 52,436 sharable pages (the page is not all zeros, and there exists at least one other identical page), and 191,646 unique pages (no other page in memory is exactly the same). The second column shows the number of pages required to store these three categories of pages using traditional page sharing. Each unique page must be preserved; however, we only need to store one copy of a set of identical pages. Hence, the 52,436 non-unique pages contain only 3,577 distinct pages — implying there are roughly fourteen copies of every non-unique page. Furthermore, only one copy of the zero page is needed, saving 149,037 pages. In total, the 393,120 original pages can be represented by 195,224 distinct pages — a 50% savings.

The third column depicts the additional savings available if we consider sub-page sharing. Using a cut-off of 2 KB for the patch size (i.e., we do not create a patch if it will take up more than half a page), we identify 144,497 distinct pages eligible for patching. We store the 50,727 remaining pages as is and use them as reference pages for the patched pages. For each of the similar pages, we compute a patch using Xdelta [76] at a low compression setting. The patches are stored in the standard VCDiff format [64] The average patch size is 1,070 bytes, allowing them to be stored in 37,695 4-KB pages, saving 106,802 pages. In sum, sub-page sharing requires only 88,422 pages to store the memory for all VMs instead of 195,224 for full-page sharing or 393,120 originally — an impressive 77% savings, or almost another 50% over full-page sharing. We note that this was the least savings in our experiments; the savings from patching are even higher in most cases. Further, a significant amount of page sharing actually comes from zero pages and, therefore, depends on their availability. For instance, the same workload when executed on 256-MB VMs yields far fewer zero pages. Alternative mechanisms to page sharing become even more important in such cases.

One of the principal complications with sub-page sharing is identifying candidate reference pages. Difference Engine uses a parametrized scheme to identify similar pages based upon the hashes of several 64-byte portions of each page. In particular, the scheme HashSimilarityDetector (k, s) hashes the contents of $(k \cdot s)$ 64-byte blocks at randomly chosen locations on the page, and then groups these hashes together into k groups of s hashes each. We use each group as an index into a hash table. In other words, higher values of s capture *local* similarity while higher k values incorporate *global* similarity. Hence, HashSimilarityDetector $(1, 1)$ will choose one block on a page and index that block; pages are considered similar if that block of data matches. HashSimilarityDetector $(1, 2)$ combines the hashes from two different locations in the page into one index of length two. HashSimilarityDetector $(2, 1)$ instead indexes each page twice: once based on the contents of a first block, and again based on the contents of a second block. Pages that match at least one of the two blocks are chosen as candidates.

For each scheme, the number of candidates, c, specifies how many different pages the hash table tracks for each signature. With one candidate, we only store the first page found with each signature; for larger values, we keep multiple pages in the hash table for each index. When trying to build a patch, Difference Engine computes a patch between all matching pages and chooses the best one. Note that since the HashSimilarityDetector(k, s) scheme looks at k hash table entries, the total number of candidates considered is actually $(k \cdot c)$.

Figure 7.2 shows the effectiveness of this scheme for various parameter settings on the two workloads described above. On the X-axis, we have parameters in the format $(k, s), c$, and on the Y-axis we plot the total savings from patching *after* all identical pages have been shared. We use the following definition of savings (we factor in the memory used to store the shared and patched/compressed pages):

$$\left(1 - \frac{\text{Total memory actually used}}{\text{Total memory allocated to VMs}}\right) \times 100$$

For both the workloads, HashSimilarityDetector $(2, 1)$ with one candidate does surprisingly well. There is a substantial gain due to hashing two distinct blocks in the page separately, but little additional gain by hashing more blocks. Combining blocks does not help much, at least for these workloads. Furthermore, storing more candidates in each hash bucket also produces little gain. Hence, Difference Engine indexes a page by hashing 64-byte blocks at two fixed locations in the page (chosen at random) and using each hash value as a separate index to store the page in the hash table. To find a candidate similar page, the system computes

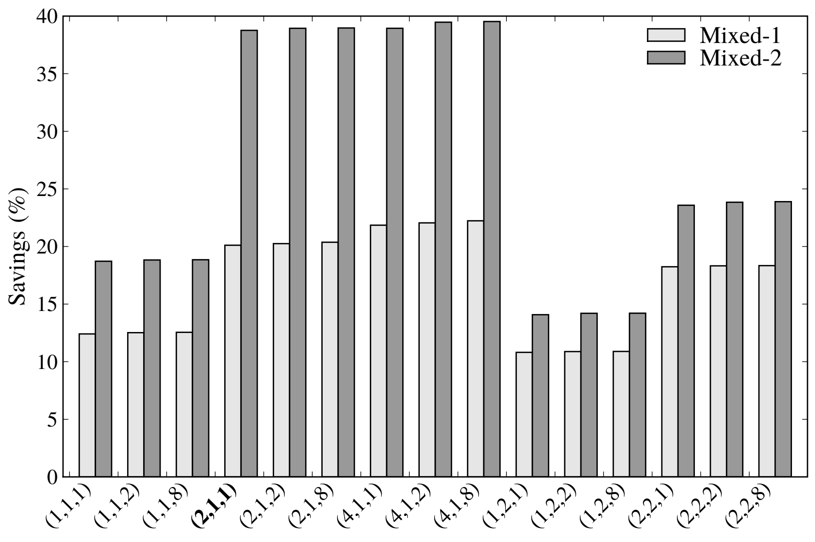


Figure 7.2: Effectiveness of the similarity detector for varying number of indices, index length and number of candidates. All entries use a 18-bit hash.

hashes at the same two locations, looks up those hash table entries, and chooses the better of the (at most) two pages found there.

Our current implementation uses 18-bit hashes to keep the hash table small to cope with the limited size of the Xen heap. In general though, larger hashes might be used for improved savings and fewer collisions. Our analysis indicates, however, that the benefits from increasing the hash size are modest. For example, using HashSimilarityDetector $(2, 1)$ with one candidate, a 32-bit hash yields a savings of 24.66% for Mixed-1, compared to a savings of 20.11% with 18-bit hashes.

7.1.3 Compression

Finally, for pages that are not significantly similar to other pages in memory, we consider compressing them to reduce the memory footprint. Compression is useful only if the compression ratio is reasonably high, and, like patching, if selected pages are accessed infrequently, otherwise the overhead of compression/decompression will outweigh the benefits. We identify candidate pages for compression using a global clock algorithm (Section 7.2.2), assuming that pages that have not been recently accessed are unlikely to be accessed in the near future.

Difference Engine supports multiple compression algorithms, currently LZO and

WKdm as described in [111]; We invalidate compressed pages in the VM and save them in a dynamically allocated storage area in machine memory. When a VM accesses a compressed page, Difference Engine decompresses the page and returns it to the VM uncompressed. It remains there until it is again considered for compression.

7.1.4 Paging Machine Memory

While Difference Engine will deliver some (typically high) level of memory savings, in the worst case our debts get called simultaneously and the VMs actually require all of their allocated memory. Setting aside sufficient physical memory to account for this case prevents using the memory saved by Difference Engine to create additional VMs. Not doing so, however, may result in temporarily overshooting the physical memory capacity of the machine and cause a system crash. We therefore require a demand-paging mechanism to supplement main memory by writing pages out to disk in such cases.

A good candidate page for swapping out would likely not be accessed in the near future — the same requirement as compressed/patched pages. In fact, Difference Engine also considers compressed and patched pages as candidates for swapping out. Once the contents of the page are written to disk, the page can be reclaimed. When a VM accesses a swapped out page, Difference Engine fetches it from disk and copies the contents into a newly allocated page that is mapped appropriately in the VM's memory.

Since disk I/O is involved, swapping in/out is an expensive operation. Further, a swapped page is unavailable for sharing or as a reference page for patching. Therefore, swapping should be an infrequent operation. Difference Engine implements the core mechanisms for paging, and leaves policy decisions — such as when and how much to swap — to user-space tools. We describe our reference implementation for swapping and the associated tools in Section 7.2.6.

Figure 7.3 summarizes the different states a page can be in and how they are related. We now describe our implementation based on the Xen VMM.

7.2 Implementation

We have implemented Difference Engine on top of Xen 3.0.4 in roughly 14,500 lines of code. An additional 20,000 lines come from ports of existing patching and compression algorithms (Xdelta, LZO, WKdm) to run inside Xen. Another 2,500 lines derive from the work

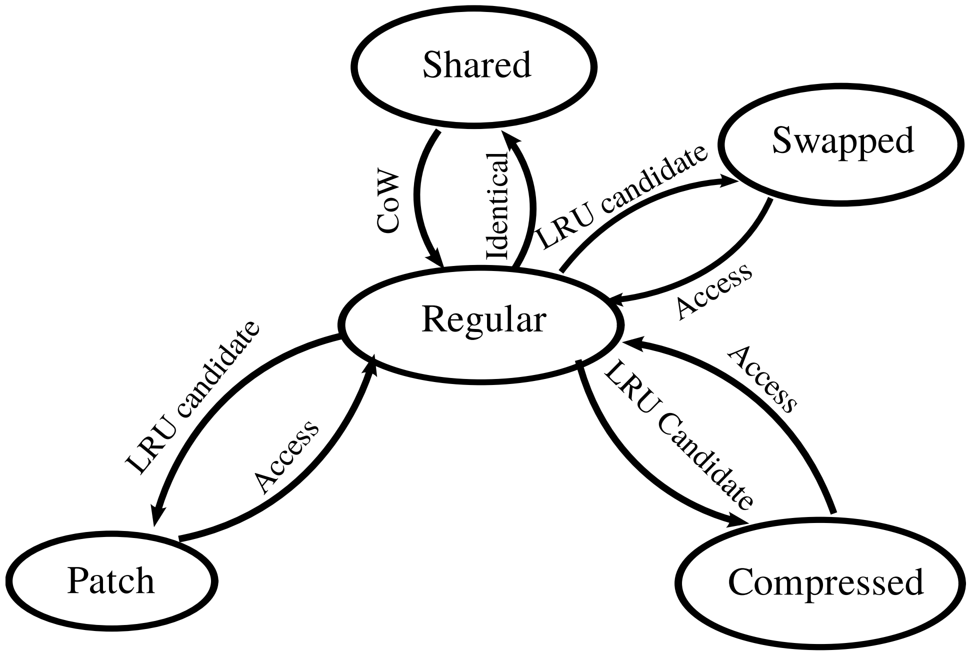


Figure 7.3: The page-state transition diagram.

Table 7.2: Memory consumed by Difference Engine data structures for identifying memory sharing opportunities. These are the fixed costs only, and do not include per-VM overhead for data structures such as shadow page tables. All memory is allocated from Xen's heap, which is under 12 MB in size.

Size	Component
1.76 MB	Page sharing hash table
1.00 MB	Page similarity hash table

by Kloster et al. [68] to implement page sharing. Modifications outside the Xen hypervisor are tiny in comparison: around 130 modified lines in `ioemu` and control tools in Domain-0, and no modifications required to the Linux kernel or elsewhere.

7.2.1 Modifications to Xen

Difference Engine relies on manipulating P2M maps and the shadow page tables to interpose on page accesses. For simplicity, we do not consider any pages mapped by Domain-0 (the privileged, control domain in Xen), which, among other things, avoids the potential for circular page faults. Our implementation method gives rise to two slight complications.

Real Mode

On x86 hardware, when the OS starts booting on bare metal, the x86 real-mode paging is disabled. This configuration is required because the OS needs to obtain some information from the BIOS for the boot sequence to proceed. When executing under Xen, this requirement means that paging is disabled during the initial stages of the boot process, and shadow page tables are not used until paging is turned on. Instead, the guest employs a direct P2M map as the page table. Hence, a VM's memory is not available for consideration by DIFFERENCE ENGINE until paging has been turned on within the guest OS.

I/O Support

To support unmodified operating system requirements for I/O access, the Xen hypervisor must emulate much of the underlying hardware that the OS expects (such as the BIOS and the display device). Xen has a software I/O emulator based on Qemu [28]. A per-VM user-space process in Domain-0 known as `ioemu` performs all necessary I/O emulation. The `ioemu` must be able to read and write directly into the guest memory, primarily for efficiency. For instance, this enables the `ioemu` process to DMA directly into pages of the VM, instead of having to DMA locally and then copying data to the VM or flipping ownership of the DMA'd pages to the VM. By virtue of executing in Domain-0, the `ioemu` may map any pages of the guest OS in its address space.

By default, `ioemu` maps the entire memory of the guest into its address space for simplicity. Recall, however, that DIFFERENCE ENGINE explicitly excludes pages mapped by Domain-0. Thus, `ioemu` will nominally prevent us from saving any memory at all, since every VM's address space will be mapped by its `ioemu` into Domain-0. Our initial prototype addressed this issue by modifying `ioemu` to map a small, fixed number (16) of pages from each VM at any given time. While simple to implement, this scheme suffered from the drawback that, for I/O-intensive workloads, the `ioemu` process would constantly have to map VM pages into its address space on demand, leading to undesirable performance degradation. To address this limitation, we implemented a dynamic aging mechanism in `ioemu` — VM pages are mapped into Domain-0 on demand, but not immediately unmapped. Every ten seconds, we unmap VM pages which were not accessed during the previous interval. To measure the real-world impact of our modification, we measured the throughout of a 580-MB `scp` transfer between two VMs. Table 7.3 shows that our current implementation is almost as fast as the original implementation, while making VM pages available for processing by the DIFFERENCE

Table 7.3: Throughout of scp between two VMs for various ioemu mapping strategies.

Mapping strategy	**Throughput**
Original (all pages)	2.84 MB/s
16 pages at a time	0.91 MB/s
Aging	2.75 MB/s
Difference Engine (dynamic aging)	2.47 MB/s

Engine.

Block Allocator

Patching and compression may result in compact representations of a page that are much smaller than the page size. We wrote a custom block allocator for Difference Engine to efficiently manage storage for patched and compressed pages. Our block allocator is loosely modeled after the slab allocator [29]. The allocator acquires pages from the domain heap (from which memory for new VMs is allocated) on demand, and returns pages to the heap when no longer required.

When the VMM starts, all of the pages in memory are regular pages. As time progresses, pages might be chosen for sharing or compression or patching. The block allocator does not acquire any memory until the first page is selected by the global clock for processing.

The storage backend grows dynamically on demand. The allocator acquires memory at page granularity; we divide each newly created page into fixed size blocks. The block size of a particular page is determined by the request that led to that page's creation. The allocator searches through the storage area for any page for that block size with a free block; if there is no such page, it allocates a new page with that block size. A nice side effect of this scheme is that there is no need to estimate or predict how many pages of each block size the allocator should maintain. This mechanism naturally converges to the actual distribution.

To minimize internal fragmentation, the block allocator manages blocks of various sizes. Block sizes are powers of two, ranging from 64 bytes to 2048 bytes (half a page). Each block stores 16 bytes of meta data, including the page type (patched or compressed), owner domain and page frame number. Each acquired page is divided into equal sized blocks. To keep track of pages containing blocks of a given size, we create stub domains, each of which owns pages of a particular block size. These stub domains are just container structures and are not scheduled by the VMM.

When the allocator receives a request for a block of a given size, it examines the last page of the corresponding stub domain. If the page is full, or if the domain owns no pages, a new page is allocated to it from the domain heap. Otherwise, the next free block found is returned to the callee. We also keep track of the number of used blocks in each page. When this drop to zero, the page can be returned to the domain heap. A nice side effect of this scheme is that we eliminate the need to estimate or predict how many pages of each block size the allocator should maintain. This mechanism naturally converges to the actual distribution.

If a free block is found, it is returned to the callee. Each page has some associated meta-data to keep track of free blocks. In particular, we leverage the type count associated with each page in the `page_info` data structure to store the number of used blocks in a page. Once the type count drops to zero, the page can be freed from the storage back end and returned to domain heap.

To find a free block within a page we perform a linear scan. To keep track of pages of different block sizes, we utilize existing infrastructure in Xen. In particular, we create stub domains, each of which owns pages of a particular block size. These stub domains are just containers for data structures and are not really scheduled as real VMs by the system. To find pages of a particular block size, we just scan the pages belonging to the stub domain for that particular page size.

7.2.2 Clock

Difference Engine implements a not recently used (NRU) policy [99] to select candidate pages for sharing, patching, compression and swapping out. On each invocation, the clock scans a portion of the memory, checking and clearing the *referenced* (R) and *modified* (M) bits on pages. Thus pages with the R/M bits set must have been referenced/modified since the last scan. The rate at which memory we scan memory is a configurable parameter. We ensure that successive scans of memory are separated by at least four seconds in the current implementation to give domains a chance to reset the R/M bits on frequently accessed pages. In the presence of multiple VMs, the clock scans a small portion of each VM's memory in turn for fairness. The interface exported by the clock is simple: return a list of pages (of some maximum size) that have not been accessed in some time.

In OSes running on bare metal, the R/M bits on page table entries are typically updated by the processor. Xen structures the P2M map exactly like the page tables used by the hardware, however since the processor doesn't actually use the P2M map as a page table, the R/M bits are not updated automatically. Xen structures the P2M map exactly like the page tables used by the hardware[1]. Since the processor does not actually use the P2M as a page table, the accessed and dirty bits are not automatically updated. We modify Xen's shadow page table code to set these bits when creating readable or writable page mappings. Unlike conventional operating systems, where there may be multiple sets of page tables that refer to the same set of pages, in Xen there is only one P2M map per domain, and so each guest page corresponds unambiguously to one P2M entry and one set of R/M bits.

Using the R/M bits, we can annotate each page with its "freshness":

- **Recently modified (C1):** The page has been written to since the last scan. [M,R=1,1]
- **Not recently modified (C2):** The page has been accessed since the last scan, but not modified. [M,R=1,0]
- **Not recently accessed (C3):** The page has not been accessed at all since the last scan. [M,R=0,0]
- **Not accessed for an extended period (C4):** The page has not been accessed in the past few scans.

Note that the existing two R/M bits are not sufficient to classify C4 pages — we extend the clock's "memory" by leveraging two additional bits in the page table entries to identify such pages. These bits are updated when a page is classified as C3 in consecutive scans. Together, these four annotations enable a clean separation between mechanism and policy, allowing us to explore different points in the policy space. The default policy of DIFFERENCE ENGINE operates as follows. C1 pages are ignored; C2 pages are considered for sharing and to be reference pages for patching, but can not be patched or compressed themselves; C3 pages can be shared or patched; C4 pages are eligible for everything, including compression and swapping.

We consider sharing first since it delivers the most memory savings in exchange for a small amount of meta data. We consider compression last because once a page is compressed, there is no opportunity to benefit from future sharing or patching of that page. An alternate, more aggressive policy might treat all pages as if they were in state C4 (not accessed in a long time) — in other words, proactively patch and compress pages. Initial experimentation indicates that while the contribution of patched and compressed pages does increase slightly,

[1]The P2M table is almost never directly used by the hardware. However, when a guest runs with paging disabled, the appropriate semantics can be achieved by setting the hardware's page table base register to point at the P2M table. So as not to confuse the hardware, we use the same bit positions for our dirty and accessed bits as the hardware page tables, even though the majority of the time these bits will not be updated by the hardware since loading the P2M as the page table will be rare.

it does not yield a significant net savings. We also considered a policy that selects pages for compression before patching. Initial experimentation with the workloads in Section 8.4.2 shows that this policy performs slightly worse than the default in terms of savings, but incurs less performance overhead since patching is more resource intensive. We suspect that it may be a good candidate policy for heterogeneous workloads with infrequently changing working sets, but do not explore it further here.

7.2.3 Page Sharing

Difference Engine uses the SuperFastHash [60] function to compute digests for each scanned page and inserts them along with the page frame number into a hash table. The implementation of this code is based upon the code of Kloster et al. [68]. Ideally, the hash table should be sized so that it can hold entries for all of physical memory. The hash table is allocated out of Xen's heap space, which is quite limited in size: the code, data, and heap segments in Xen must all fit in a 12-MB region of memory. Changing the heap size requires pervasive code changes in Xen, and will likely break the application binary interface (ABI) for some OSes. We therefore restrict the size of the page-sharing hash table so that it can hold entries for only $1/5$ of physical memory. Hence Difference Engine processes memory in five passes (we refer the reader to [68] for details). In our test configuration, this partitioning results in a 1.76-MB hash table. We divide the space of hash function values into five intervals, and only insert a page into the table if its hash value falls into the current interval. A complete cycle of five passes covering all the hash value intervals is required to identify all identical pages.

7.2.4 Page Similarity Detection

The goal of the page similarity component is to find pairs of pages with similar content, and, hence, make candidates for patching. We implement a simple strategy for finding similar pages based on hashing short blocks within a page, as described in Section 7.1.2. Specifically, we use the HashSimilarityDetector $(2, 1)$ described there, which hashes short data blocks from two locations on each page, and indexes the page at each of those two locations in a separate page similarity hash table, distinct from the page sharing hash table described above. We use the 1-candidate variation, where at most one page is indexed for each block hash value.

Recall that the clock makes a complete scan through memory in five passes. The page sharing hash table is cleared after each pass, since only pages *within* a pass are considered for sharing. However, two similar pages may appear in different passes if their hash values fall in different intervals. Since we want to only consider pages that have not been shared in a full cycle for patching, the page similarity hash table is *not* cleared on every pass. This approach also increases the chances of finding better candidate pages to act as the reference for a patch.

The page similarity hash table *may* be cleared after considering every page in memory — that is, at the end of each cycle of the global clock. We do so to prevent stale data from accumulating: if a page changes after it has been indexed, we should remove old pointers to it. Since we do not trap on write operations, it is simpler to just discard and rebuild the similarity hash table.

Only the last step of patching — building the patch and replacing the page with it — requires a lock. We perform all earlier steps (indexing and lookups to find similar pages) without pausing any domains. Thus, the page contents may change after DIFFERENCE ENGINE indexes the page, or after it makes an initial estimate of patch size. This situation is fine since the goal of these steps is to find pairs of pages that will likely patch well. An intervening page modification will not cause a correctness problem, only a patch that is larger than originally intended.

7.2.5 Compression

Compression operates similarly to patching — in both cases the goal is to replace a page with a shorter representation of the same data. The primary difference is that patching makes use of a reference page, while a compressed representation is self-contained.

There is one important interaction between compression and patching: once we compress a page, the page can no longer be used as a reference for a later patched page. A naive implementation that compresses all non-identical pages as it goes along will almost entirely prevent page patches from being built. Compression of a page should be postponed at least until all pages have been checked for similarity against it. A complete cycle of a page sharing scan will identify similar pages, so a sufficient condition for compression is that *no page should be compressed until a complete cycle of the page sharing code finishes*. We make the definition of “not accessed for an extended period” in the clock algorithm coincide with this condition (state C4). As mentioned in Section 7.2.2, this is our default policy for page compression.

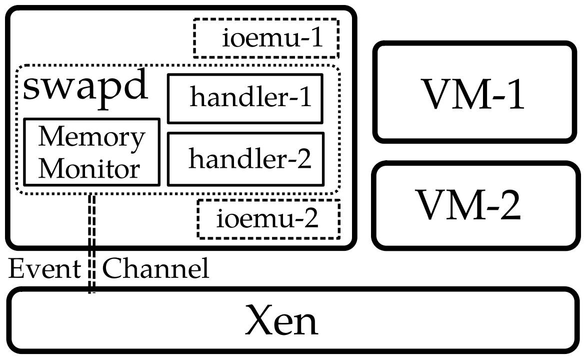


Figure 7.4: Architecture of the swap mechanism.

7.2.6 Paging Machine Memory

Recall that any memory freed by Difference Engine cannot be used reliably without supplementing main memory by secondary storage. That is, when the total allocated memory of all VMs exceeds the system memory capacity, some pages will have to be swapped to disk. Note that this ability to over-commit memory is useful in Xen independent of other Difference Engine functionality, and has been designed accordingly.

Since the Xen VMM does not handle any I/O (all I/O is delegated to Domain-0) and is not aware of any devices, it is not possible to build swap support directly in the hypervisor. Further, since Difference Engine supports unmodified OSes, we cannot expect any support from the guest OS. Figure 7.4 shows the design of our swap implementation guided by these constraints. A single Swap Daemon (`swapd`) running as a user process in Domain-0 manages the swap space. For each VM in the system, `swapd` creates a separate thread to handle swap-in requests. Swapping out is initiated by `swapd`, when one of the following occurs:

- the memory utilization in the system exceeds some user configurable threshold (the `HIGH_WATERMARK`). Pages are swapped out until a user configurable threshold of free memory is attained (the `LOW_WATERMARK`). A separate thread (the `memory_monitor`) tracks system memory.

- a swap out notification is received from Xen via an event channel. This allows the hypervisor to initiate swapping if more memory is urgently required (for instance, when creating a private copy of a shared page). The hypervisor indicates the amount of free memory desired.

- a swap out request is received from another process. This allows other user-space tools (for instance, the VM creation tool) to initiate swapping in order to free memory. We currently employ XenStore [24] for such communication, but any other IPC mechanism can be used.

Note that swapd always treats a swap out request as a hint. It will try to free pages, but if that is not possible — if no suitable candidate page was available, for instance, or if the swap space became full — it continues silently. A single flat file of configurable size is used as the back-end storage for the swap space.

To swap out a page, swapd makes a hypercall into Xen, where a victim page is chosen by invoking the global clock. If the victim is a compressed or patched page, we first reconstruct it. We pause the VM that owns the page and copy the contents of the page to a page in Domain-0's address space (supplied by swapd). Next, we remove all entries pointing to the victim page in the P2M and M2P maps, and in the shadow page tables, and mark the page as swapped out in the corresponding page table entry. Meanwhile, swapd writes the page contents to the swap file and inserts the corresponding byte offset in a hash table keyed by <Domain ID, guest page frame number>. Finally, we free the page, return it to the domain heap, and reschedule the VM.

When a VM tries to access a swapped page, it incurs a page fault and traps into Xen. We pause the VM and allocate a fresh page to hold the swapped in data. We populate the P2M and M2P maps appropriately to accommodate the new page. Xen dispatches a swap-in request to swapd containing the domain ID and the faulting page frame number. The handler thread for the faulting domain in swapd receives the request and fetches the location of the page in the swap file from the hash table. It then copies the page contents into the newly allocated page frame within Xen via another hypercall. At this point, swapd notifies Xen, and Xen restarts the VM at the faulting instruction.

This implementation leads to two interesting interactions between ioemu and swapd. First, recall that ioemu can directly write to a VM page mapped in its address space. Mapped pages might not be accessed until later, so a swapped page can get mapped or a mapped page can get swapped out without immediate detection. To avoid unnecessary subsequent swap ins, we modify ioemu to ensure that pages to be mapped will be first swapped in if necessary and that mapped pages become ineligible for swapping. Also note that control must be transferred from Xen to swapd for a swap in to complete. This asynchrony allows a race condition where ioemu tries to map a swapped out page (so Xen initiates a swap in on its behest) and proceeds with the access before the swap in has finished. This race can happen because both processes must run in Domain-0 in the Xen architecture. As a work around, we modify `ioemu` to block if a swap in is still in progress inside `swapd` using shared memory between the processes for the required synchronization.

While implementing swap support in Xen was a necessity for Difference Engine and a valuable learning experience, we are nonetheless inclined to conclude that the current I/O architecture for fully-virtualized domains is not amenable for an efficient swap implementation. Ideally, swapping in should be an *atomic* process, not an asynchronous one. Making all the necessary modifications was difficult and likely brittle to updates to the underlying Xen code.

Chapter 7, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI) 2008. Gupta, Diwaker; Lee, Sangmin; Vrable, Michael; Savage, Stefan; Snoeren, Alex C.; Varghese, George; Voelker, Geoffrey M.; Vahdat, Amin. The dissertation author is the primary investigator and author of this paper.

Chapter 8

Difference Engine Evaluation

We first present micro-benchmarks to evaluate the cost of individual operations and the performance of the global clock. We then consider each mechanism in isolation to gain a better understanding of the trade-offs involved. Next, we evaluate whole-system performance: for a range of workloads, we measure memory savings and the impact on application performance. We quantify the contributions of each DIFFERENCE ENGINE mechanism, and also present head-to-head comparisons with the VMware ESX server. We consider several different configurations to quantify the memory savings when subjected to a broad range of workload mixes. Finally, we demonstrate how our memory savings can be used to boost the aggregate system performance. Unless otherwise mentioned, all experiments are run on dual-processor, dual-core 2.33-GHz Intel Xeon machines and the page size is 4 KB.

8.1 Cost of Individual Operations

Before quantifying the memory savings provide by DIFFERENCE ENGINE, we measure the overhead of various functions involved. We obtain these numbers by enabling each mechanism in isolation (thus if page sharing is being benchmarked, patching and compression are disabled), and running the custom micro-benchmark described in Section 8.3. To benchmark paging, we disabled all three mechanisms and forced eviction of 10,000 pages from a single 512-MB VM. We then ran a simple program in the VM that touches all memory to force pages to be swapped in.

Table 8.1 shows the overhead imposed by the major DIFFERENCE ENGINE operations. As expected, collapsing identical pages into a copy-on-write shared page (`share_page`) and

Table 8.1: CPU overhead of different functions.

Function	**Mean execution time (μs)**
`share_pages`	6.2
`cow_break`	25.1
`compress_page`	29.7
`uncompress`	10.4
`patch_page`	338.1
`unpatch`	18.6
`swap_out_page`	48.9
`swap_in_page`	7151.6

recreating private copies (`cow_break`) are relatively cheap operations, taking approximately 6 and 25 μs, respectively. Perhaps more surprising, however, is that compressing a page on our hardware is fast, requiring slightly less than 30 μs on average. Patching, on the other hand, is almost an order of magnitude slower: creating a patch (`patch_page`) takes over 300 μs. This time is primarily due to the overhead of finding a good candidate base page and constructing the patch. Both decompressing a page and re-constructing a patched page are also fairly fast, taking 10 and 18 μs respectively.

Swapping out takes approximately 50 μs. However, this does *not* include the time to actually write the page to disk. This is intentional: once the page contents have been copied to user-space, they *are* immediately available for being swapped in; and the actual write to the disk might be delayed because of file system and OS buffering in Domain-0. Swapping in, on the other hand, is the most expensive operation, taking approximately 7 ms. There are a few caveats, however. First, swapping in is an asynchronous operation and might be affected by several factors, including process scheduling within Domain-0; it is *not* a tight bound. Second, swapping in might require reading the swapped out page from disk, and the seek time will depend on the size of the swap file, among other things.

8.2 Clock Performance

The performance of applications running with Difference Engine depends upon how effectively we choose idle pages to compress or patch. Patching and compression are computationally-intensive, and the benefits of this overhead last only until the next access to the page. Reads are free for shared pages, but not so for compressed or patched pages. The clock algorithm is intended to only consider pages for compression/patching that are not likely

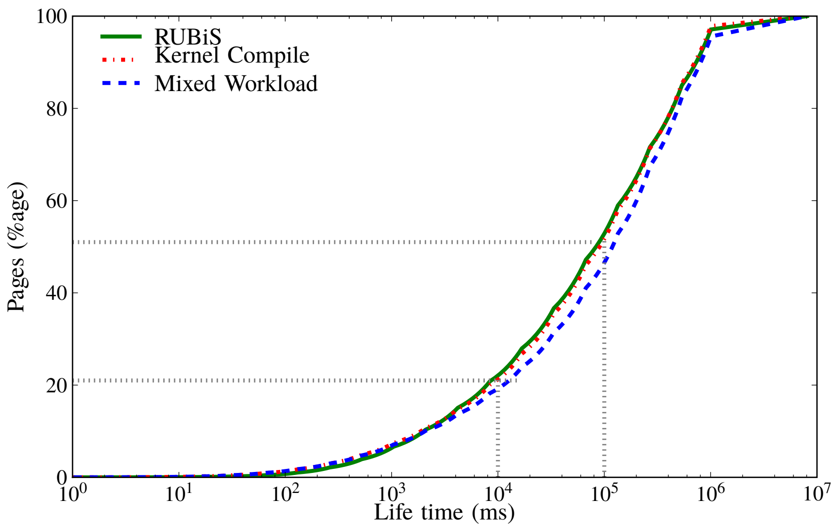
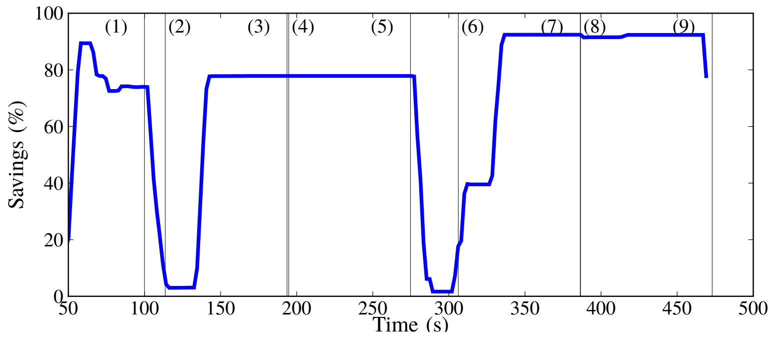


Figure 8.1: Lifetime of patched and compressed pages for three different workloads. Our NRU implementation works well in practice.

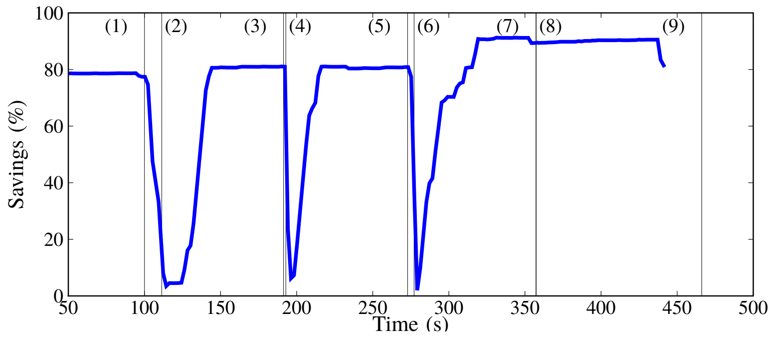
to be accessed again soon; here we evaluate how well it achieves that goal.

For three different workloads, we trace the lifetime of each patched and compressed page. The lifetime of a page is the time between when it was patched/compressed, and the time of the first subsequent access (read or write). The workloads range from best case homogeneous configurations (same OS, same applications) to a worst case, highly heterogeneous mix (different OSes, different applications). The RUBiS and kernel compile workloads use four VMs each (Section 8.4.1). We use the Mixed-1 workload described earlier (Section 7.1.2) as the heterogeneous workload.

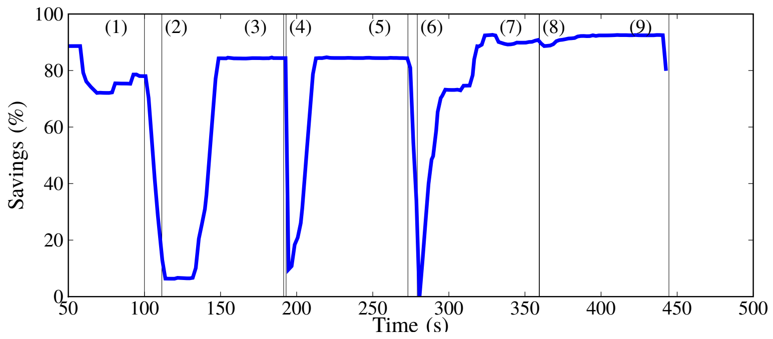
Figure 8.1 plots the cumulative distribution of the lifetime of a page: the X-axis shows the lifetime (in ms) in log scale, and the Y-axis shows the fraction of compressed/patched pages. A good clock algorithm should give us high lifetimes, since we would like to patch/compress only those pages which will not be accessed in the near future. As the figure shows, almost 80% of the victim pages have a lifetime of at least 10 seconds, and roughly 50% have a lifetime greater than 100 seconds. This is true for both the homogeneous and the mixed workloads, indicating that our NRU implementation works well in practice.



(a) Sharing

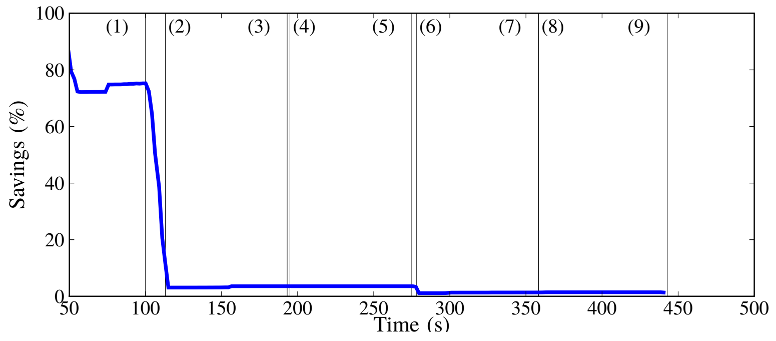


(b) Patching

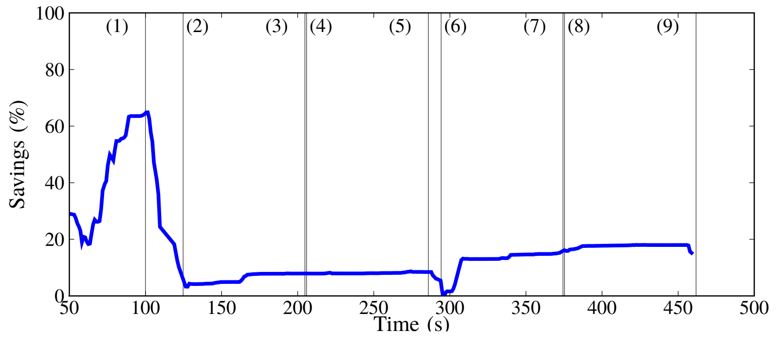


(c) Compression

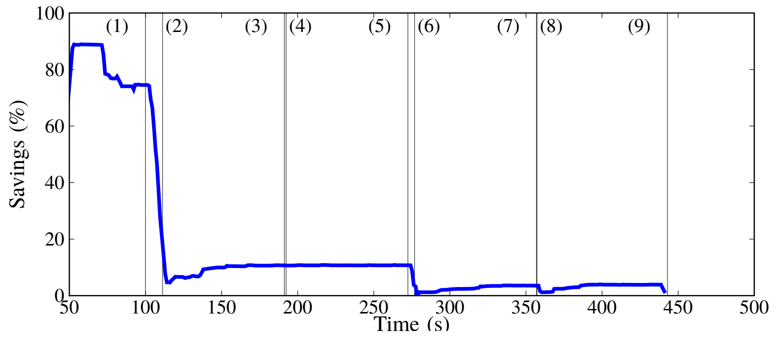
Figure 8.2: Workload: Identical pages. Performance with zero pages is very similar. All mechanisms exhibit similar gains.



(a) Sharing

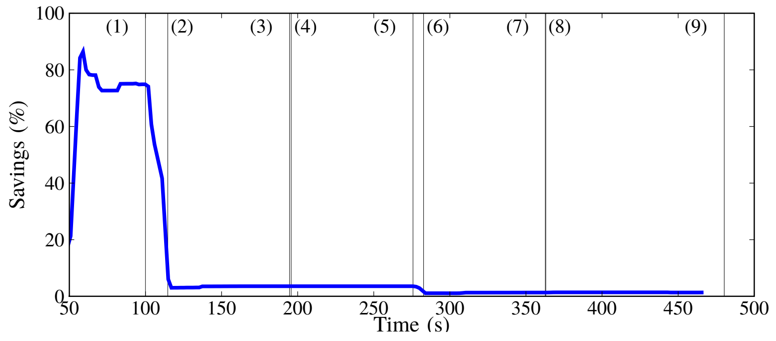


(b) Patching

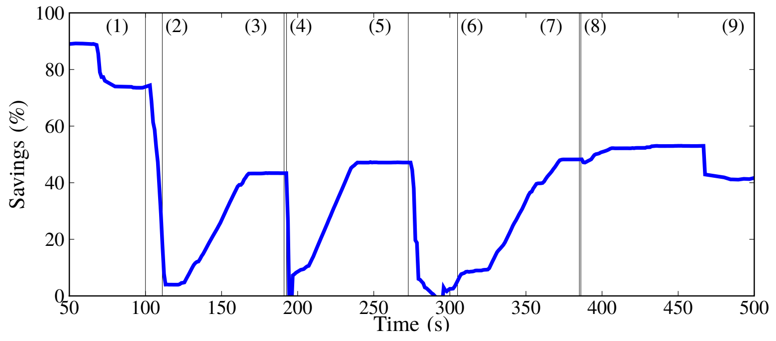


(c) Compression

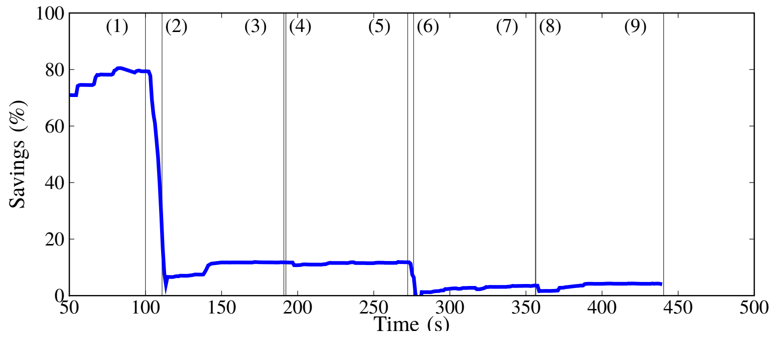
Figure 8.3: Workload: Random pages. None of the mechanisms perform very well, with sharing saving the least memory.



(a) Sharing



(b) Patching



(c) Compression

Figure 8.4: Workload: Similar pages with 95% similarity. Patching does significantly better than compression and sharing.

8.3 Techniques in Isolation

To understand the individual contribution of the three techniques, we first quantify the performance of each Difference Engine mechanism in isolation. We deployed Difference Engine on three machines running Debian 3.1 on a VM. Each machine is configured to use a single mechanism—one machine uses just page sharing, one uses just compression, and one just patching. We then subject all the machines to the same workload and profile the memory utilization.

To help distinguish the applicability of each technique to various page contents, we choose a custom workload generator that manipulates memory in a repeatable, predictable manner over off-the-shelf benchmarks. This enables fairer comparison across different application/OS mixes, as well as normalizes against any benchmark specific peculiarities. Further, most benchmarks are designed to test specific sub systems or applications (web server, database etc) and so are not particularly suited for our micro-benchmarks. Our workload generator runs in four phases. First it allocates pages of a certain type. To exercise the different mechanisms in predictable ways, we consider four distinct page types: zero pages, random pages, identical pages and similar-but-not-identical pages. Second, it reads all the allocated pages. Third, it makes several small writes to all the pages. Finally, it frees all allocated pages and exits. After each step, the workload generator idles for some time, allowing the memory to stabilize. For each run of the benchmark, we spawn a new VM and start the workload generator within it. At the end of each run, we destroy the container VM and again give memory some time to stabilize before the next run. We ran benchmarks with varying degrees of similarity, where similarity is defined as follows: a similarity of 90% means all pages differ from a base page by 10%, and so any two pages will differ from each other by at most 20%. Here, we present the results for 95%-similar pages, but the results for other values are similar.

Each VM image is configured with 256 MB of memory. Our workload generator allocates pages filling 75% (192 MB) of the VM's memory. The stabilization period is a function of several factors, particularly the period of the global clock. For these experiments, we used a sleep time of 80 seconds between each phase. During the write step, the workload generator writes a single constant byte at 16 fixed offsets in the page. As a result, identical pages remain identical after the write phase. For each configuration, we monitor the memory savings over time as the run proceeds. On each of the time series graphs, the significant events during the run are marked with a vertical line. These events are: (1) begin and (2) end of the allocation phase, (3) begin and (4) end of the read phase, (5) begin and (6) end of the write phase, (7) begin and (8) end of the free phase, and (9) VM destruction.

Figure 8.2 shows the memory savings as a function of time for each mechanism for identical pages (we omit results with zero pages – they are essentially the same as identical pages). Note that while each mechanism achieves similar savings, the crucial difference is that reads are free for page sharing. With compression/patching, even a read requires the page to be reconstructed, leading to the sharp decline in savings around event (3) and (5).

At the other extreme are random pages. Intuitively, none of the mechanisms should work very well since the opportunity to share memory is scarce. Figure 8.3 demonstrates this: once the pages have been allocated, none of the mechanisms are able to share more than 15–20% memory. Page sharing does the worst, managing a meager 5% at best.

From the perspective of page sharing, similar pages are no better than random pages. However, patching should take advantage of sub-page similarity across pages. Figure 8.4 shows the memory savings for the workload with pages of 95% similarity. Note how similar the graphs for sharing and compression look for similar and random pages. Patching, on the other hand, does substantially better, extracting up to 55% savings. Since we search for candidate pages online, often times the match found is sub optimal, which explains why patching does not do even better.

8.4 Real-world Applications

Armed with a concrete understanding of the performance of each individual technique, we now present the performance of Difference Engine on a variety of workloads. We seek to answer two questions. First, how effective are the memory-saving mechanisms at reducing memory usage for real-world applications? Second, what is the impact of those memory-sharing mechanisms on system performance? Since the degree of possible sharing depends on the software configuration, we consider several different cases of application mixes.

To put our numbers in perspective, we also do head-to-head comparisons with VMware ESX server for three different workload mixes. We run ESX Server 3.0.1 build 32039 on a Dell PowerEdge 1950 system. Note that even though this system has two 2.3-GHz Intel Xeon processors, our VMware license limits our usage to a single CPU. Hence, while doing the comparison, we restrict Xen (and, hence, Difference Engine) to use a single CPU for fairness. We also ensure that the OS images used with ESX match those used with Xen, especially the file system and disk layout. Note that we are only concerned with the effectiveness of the memory sharing mechanisms—not in comparing the application performance across the two hypervisors. Further, we configure ESX to use its most aggressive page sharing settings where it scans 10,000 pages/second (default 200); we configure Difference Engine similarly.

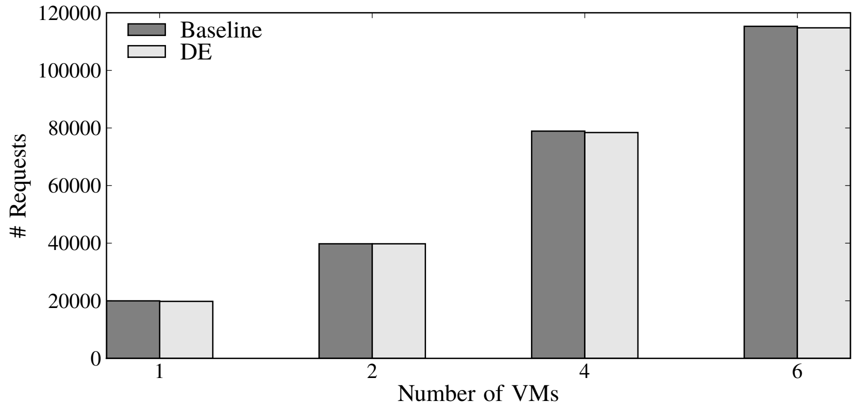
8.4.1 Base Scenario: Homogeneous VMs

In our first set of benchmarks, we test the base scenario where all VMs on a machine run the same OS and applications. This scenario is common in cluster-based systems where several services are replicated to provide fault tolerance or load balancing. Our expectation is that significant memory savings are available and that most of the savings will come from page sharing.

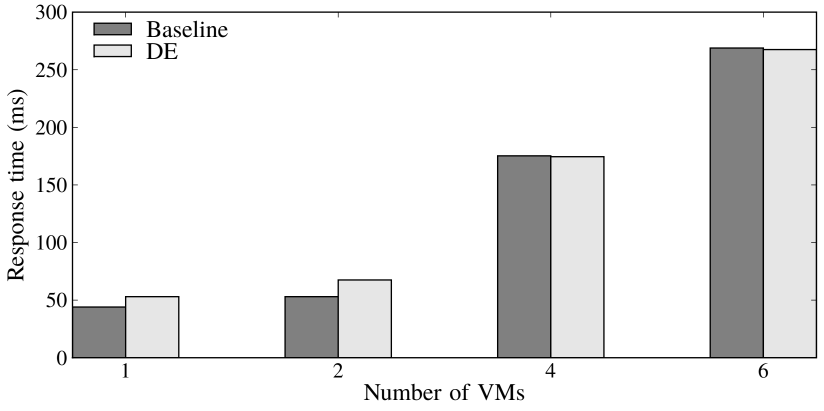
On a machine running standard Xen, we start from 1 to 6 VMs, each with 256 MB of memory and running RUBiS [37] — an e-commerce application designed to evaluate application server performance — on Debian 3.1. We use the PHP implementation of RUBiS; each instance consists of a web server (Apache) and a database server (MySQL). Two distinct client machines generate the workload, each running the standard RUBiS workload generator simulating 100 user sessions. The benchmark runs for roughly 20 minutes. The workload generator reports several metrics at the end of the benchmark, in particular the average response time and the total number of requests served.

We then run the same set of VMs with Difference Engine enabled. Figures 8.5a and 8.5b show that both the total number of requests and the average response time remain unaffected while delivering 65–75% memory savings in all cases. In Figure 8.5c, the bars indicate the average memory savings over the duration of the benchmark. Without memory sharing, memory requirements naturally grow linearly with the number of VMs. Each bar also shows the individual contribution of each mechanism. Note that in this case, the bulk of memory savings comes from page sharing. Recall that Difference Engine tries to share as many pages as it can before considering pages for patching and compression, so sharing is expected to be the largest contributor in most cases, particularly in homogeneous workloads. Interestingly, even with a single virtual machine, almost 70% memory can be saved. We emphasize that these savings are *while* the benchmark is running: on an otherwise idle system (or a stable system with little I/O activity), Difference Engine consistently achieves more than 80% savings.

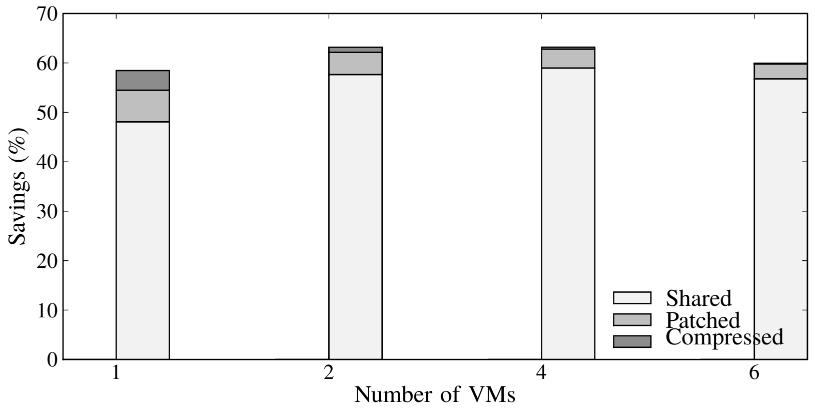
Next, we conduct a similar experiment where each VM compiles the Linux kernel (version 2.6.18). Since the working set of VMs changes much more rapidly in a kernel compile, as many files are being read and written, we expect less memory savings compared



(a) Total requests handled

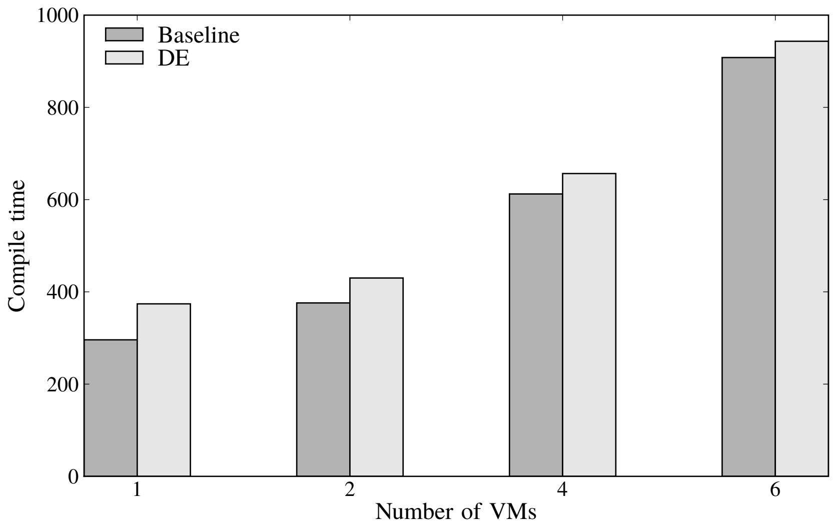


(b) Average response time

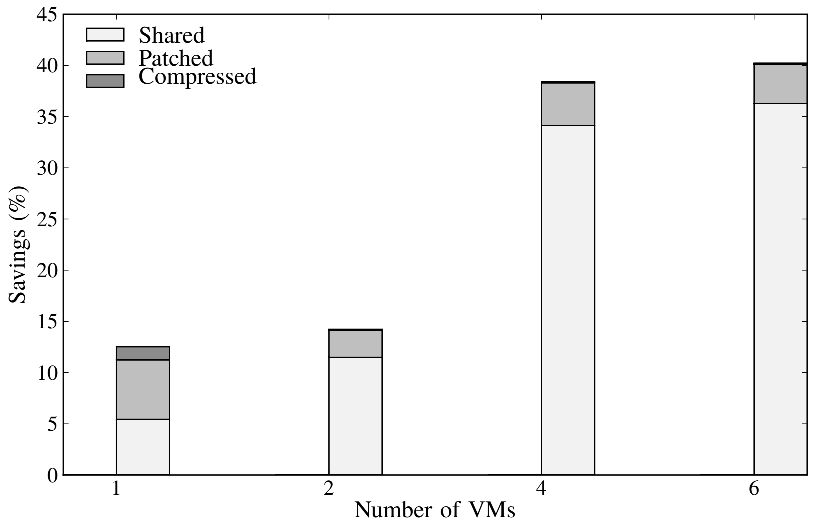


(c) Average and maximum savings

Figure 8.5: Difference Engine performance with homogeneous VMs running RUBiS



(a) Compile Time



(b) Savings

Figure 8.6: Difference Engine performance with homogeneous VMs compiling the Linux kernel.

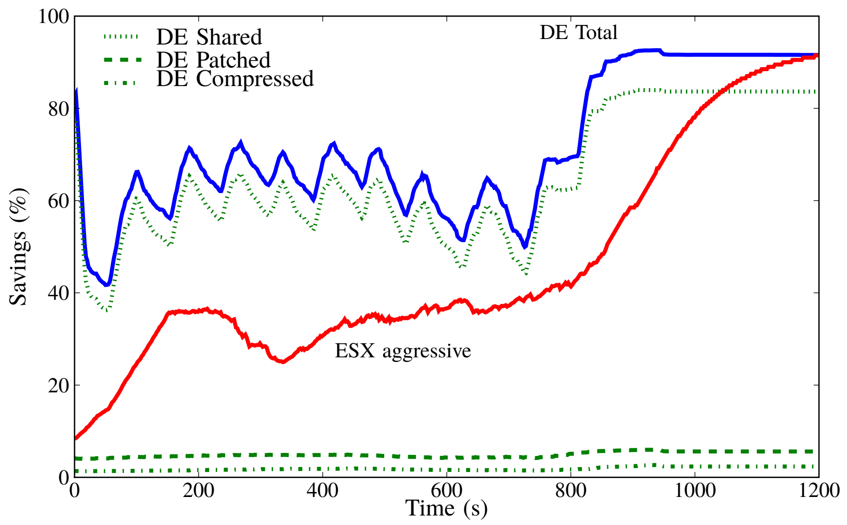


Figure 8.7: Four identical VMs executing `dbench`. For such homogeneous workloads, both Difference Engine and ESX eventually yield similar savings, but DE extracts more savings *while* the benchmark is in progress.

to the RUBiS workload. As before, we measure the time taken to finish the compile and the memory savings for varying number of virtual machines. Figures 8.6a and 8.6b plot the time taken for the kernel compile (averaged across all VMs) and the memory savings in each case. First, note that compared to RUBiS memory savings are lower, but still significant. And the amount of memory we can save increases as we increase the number of VMs. Secondly, in all cases, we incur a performance penalty of at most 5–10%. For such an I/O intensive workload, we find the memory savings (*during* the workload) impressive. Again, shared pages contribute most to the savings.

We next compare Difference Engine performance with the VMware ESX server. We set up four 512-MB virtual machines running Debian 3.1. Each VM executes dbench [98] for ten minutes followed by a stabilization period of 20 minutes. Figure 8.7 shows the amount of memory saved as a function of time. First, note that *eventually* both ESX and Difference Engine reclaim roughly the same amount of memory (the graph for ESX plateaus beyond 1200 seconds). However, *while* dbench is executing, Difference Engine delivers approximately 1.5 times the memory savings achieved by ESX. As before, the bulk of Difference Engine savings come from page sharing for the homogeneous workload case.

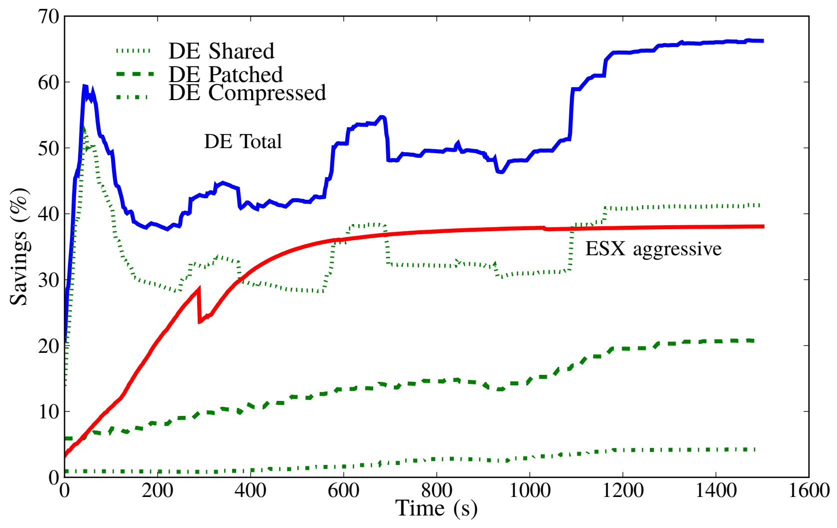


Figure 8.8: Memory savings for Mixed-1. Difference Engine saves up to 45% more memory than ESX.

8.4.2 Heterogeneous OS and Applications

Given the increasing trend towards virtualization, both on the desktop and in the data center, we envision that a single physical machine will host significantly different types of operating systems and workloads. While smarter VM placement and scheduling will mitigate some of these differences, there will still be a diverse and heterogeneous mix of applications and environments, underscoring the need for mechanisms other than page sharing. We now examine the utility of Difference Engine in such scenarios, and demonstrate that significant additional memory savings result from employing patching and compression in these settings.

Figures 8.8 and 8.9 show the memory savings as a function of time for the two heterogeneous workloads—Mixed-1 and Mixed-2 described in Section 7.1.2. We make the following observations. First, in steady state, Difference Engine delivers a factor of 1.6-2.5 more memory savings than ESX. For instance, for the Mixed-2 workload, with a factor of 2 more memory savings, Difference Engine could host the three VMs allocated 512 MB of physical memory each in approximately 760 MB of machine memory; ESX would require roughly 1100 MB of machine memory. The remaining, significant, savings come from patching and compression. And these savings come at a small cost. Table 8.2 summarizes the performance of the three benchmarks in the Mixed-1 workload. The baseline configuration is

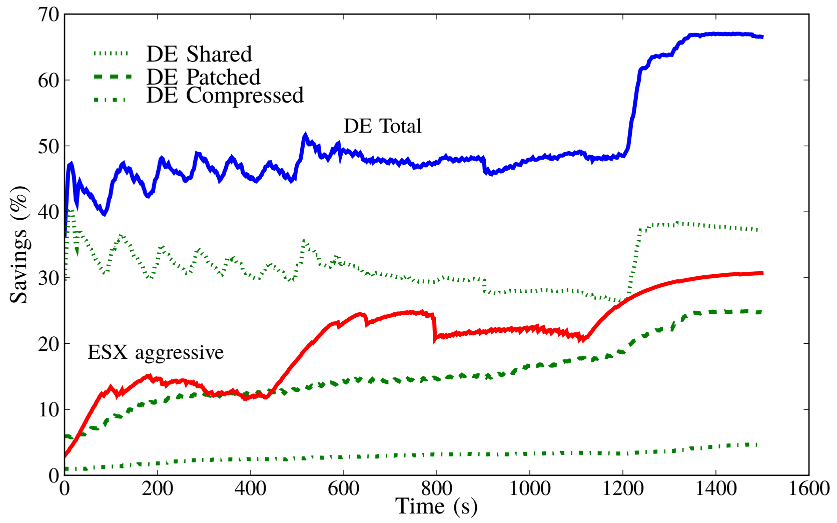


Figure 8.9: Memory savings for Mixed-2. Difference Engine saves almost twice as much memory as ESX.

Table 8.2: Application performance under Difference Engine for the heterogeneous workload Mixed-1 is within 7% of the baseline.

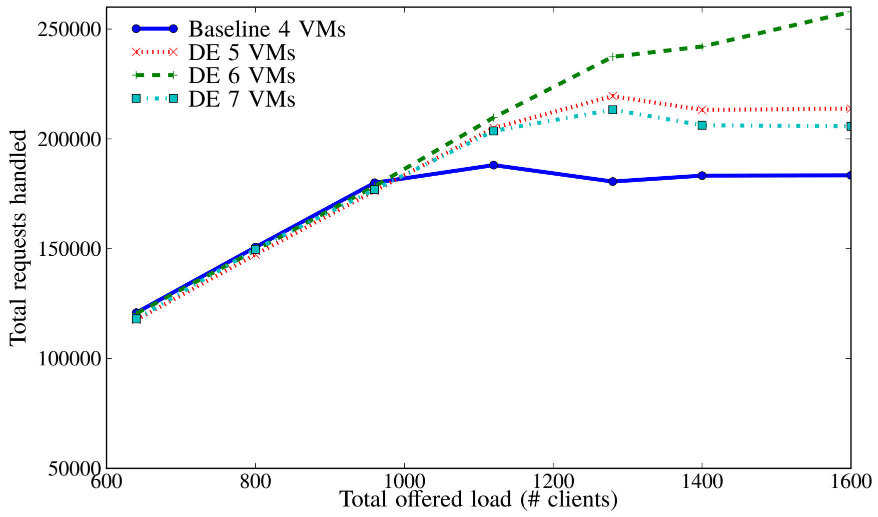
	Kernel Compile (sec)	**Vim compile, lmbench (sec)**	**RUBiS requests**	**RUBiS response time(ms)**
Baseline	670	620	3149	1280
DE	710	702	3130	1268

regular Xen without Difference Engine. In all cases, performance overhead of Difference Engine is within 7% of the baseline. For the same workload, we find that performance under ESX with aggressive page sharing is also within 5% of the ESX baseline with no page sharing.

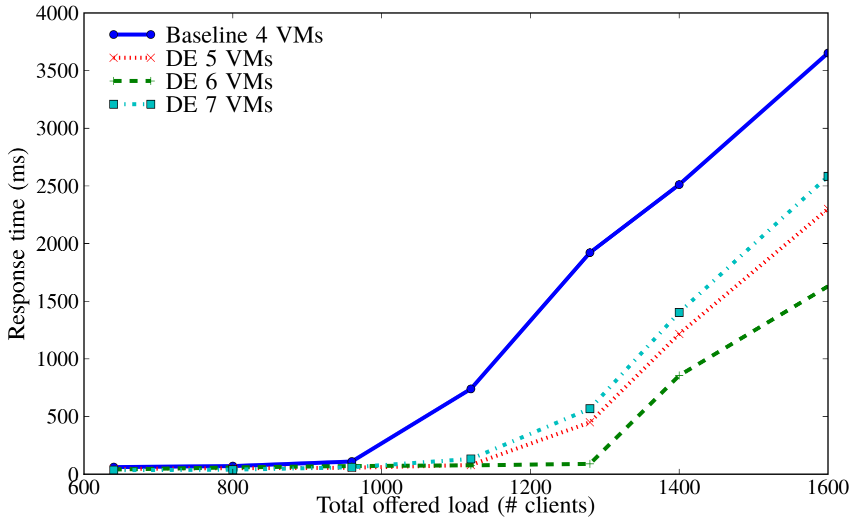
8.4.3 Increasing Aggregate System Performance

Difference Engine goes to great lengths to reclaim memory in a system, but eventually this extra memory needs to actually get used in a productive manner. One can certainly use the saved memory to create more VMs, but does that increase the aggregate system performance?

To answer this question, we created four VMs with 650 MB of RAM each on a physical machine with 2.8 GB of free memory (excluding memory allocated to Domain-



(a) Total requests handled



(b) Average response time

Figure 8.10: Up to a limit, Difference Engine can help increase aggregate system performance by spreading the load across extra VMs.

0). For the baseline (without Difference Engine), Xen allocates memory statically. Upon creating all the VMs, there is clearly not enough memory left to create another VM of the same configuration. Each VM hosts a RUBiS instance. For this experiment, we used the Java Servlets implementation of RUBiS, with Apache Tomcat as the servlet container. There are two distinct client machines per VM to act as workload generators.

The goal is to increase the load on the system to saturation. The solid lines in Figures 8.10a and 8.10b show the total requests served and the average response time for the baseline, with the total offered load marked on the X-axis. Note that beyond 960 clients, the total number of requests served plateaus at around 180,000 while the average response time increases sharply. Upon investigation, we find that for higher loads all of the VMs have more than 95% memory utilization and some VMs actually start swapping to disk (within the guest OS). Using fewer VMs with more memory (for example, 2 VMs with 1.2 GB RAM each) did not improve the baseline performance for this workload.

Next, we repeat the same experiment with Difference Engine, except this time we utilize reclaimed memory to create additional VMs. As a result, for each data point on the X-axis, the per VM load decreases, while the aggregate offered load remains the same. We expect that since each VM individually has lower load compared to the baseline, the system will deliver better aggregate performance. The remaining lines in Figures 8.10a and 8.10b show the performance with up to three extra VMs. Clearly, Difference Engine enables higher aggregate performance and better response time compared to the baseline. However, beyond a certain point (two additional VMs in this case), the overhead of managing the extra VMs begins to offset the performance benefits: Difference Engine has to effectively manage 4.5 GB memory on a system with 2.8 GB RAM to support 7 VMs. In each case, beyond 1400 clients, the VMs working set becomes large enough to invoke the paging mechanism: we observe between 5000 pages (for one extra VM) to around 20,000 pages (for three extra VMs) being swapped out, of which roughly a fourth get swapped back in.

We expect that in general Difference Engine will improve system utilization and aggregate performance for many workloads; validating this on other applications is part of future work. In particular, a tool like DieCast would clearly benefit from Difference Engine: the efficiency of a DieCast-scaled system depends on the multiplexing factor used. Difference Engine can deliver higher multiplexing factors without sacrificing performance. In fact, Difference Engine is even better suited to run under time dilation, since any overheads due to the mechanisms of Difference Engine would be masked by the slow time-frame within the guest OS.

Chapter 8, in part, is a reprint of the material as it appears in Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI) 2008. Gupta, Diwaker; Lee, Sangmin; Vrable, Michael; Savage, Stefan; Snoeren, Alex C.; Varghese, George; Voelker, Geoffrey M.; Vahdat, Amin. The dissertation author is the primary investigator and author of this paper.

Chapter 9

Conclusions

Virtualization offers a potent demonstration of two broad principles of computer science: better abstractions and layers of indirection. A virtual machine provides a software abstraction for a real physical machine, effectively providing an application-like higher level interface. This compact, isolated abstraction enables several new and interesting applications. Likewise, by decoupling the real hardware from the machine interface visible to the OS, virtualization provides increased flexibility in provisioning resources.

Recent years have witnessed an explosive growth in the interest and use of virtualization. The adoption of virtual machines is motivated primarily by server consolidation: by aggregating multiple services on a fewer physical machines, organizations can reduce power and cooling costs and increase resource utilization levels. However, virtualization is becoming increasingly more compelling for the new and interesting applications it enables, in security, intrusion detection, testing and development, backup and disaster recovery and so on.

The central thesis of this dissertation is that the ability to efficiently multiplex VMs is critical to realizing many of the benefits of virtualization — both for supporting existing applications such as server consolidation, and for enabling new classes of applications. As we discussed in Chapter 1, conventional mechanisms of resource multiplexing lead to resource partitioning: VMs on the same physical machine share the underlying resources. This limits not only the number of VMs that can be supported on a single physical machine (horizontal scalability), but also the resources available to individual VMs (vertical scalability).

We address the scalability of multiplexing along both these dimensions. We show that by making more efficient use of machine memory, we can improve horizontal scalability, doubling the number of supported VMs in some cases. Critically, we demonstrate these benefits at a negligible performance overhead. To address vertical scalability, we present mechanisms to increase the perceived resource capacity of individual VMs. More importantly, we show that this enables novel and interesting applications in scalable network emulation and large-scale testing. For instance, we show that is possible to evaluate protocols on a 100-Gbps wide-area link using 1-Gbps hardware. Similarly, we demonstrate that a given testing infrastructure can be used to accurately replicate and test systems that are an order of magnitude larger.

We present some directions for future research below and conclude the chapter with a discussion of the limitations of the approaches proposed in this dissertation and a summary of our contributions.

9.1 Future Work

This dissertation presents several interesting avenues for future work. We highlight the salient ones below.

9.1.1 Infrastructure Optimization

One of the challenges of using virtual machines in a data center environment is deciding how place the VMs among the available physical machines — note that this includes the initial placement when the VM is created, as well as any subsequent migrations. Several placement strategies are possible: for instance, VMs running CPU intensive workloads might be spread uniformly across the data center to minimize hotspots; alternatively, VMs might be placed for smooth heat dissipation in the data center in order to minimize the cooling costs. Similarly, a system like Difference Engine would benefit the most if similar VMs are physically co-located, where similarity is in terms of the contents of the VM's memory.

Consider how such a placement strategy might work. We want to establish the similarity of the memory contents among all pairs of VMs in the data center and use that to guide the placement and migration decisions. Bear in mind that the memory contents might be changing over time, and that each VM might be allocated gigabytes of memory. Concretely, we require:

- a compact and efficient representation of a VM's memory contents: it is infeasible to transport hashes of 1-GB worth of memory pages over the network, for instance; and
- a mechanism to quickly compare above mentioned representations to estimate similarity across VMs and use it to guide placement and migration policies.

We outline a preliminary sketch for such a scheme, based on min-wise independent permutations [31]. The key insight is that min-wise hashes can be used to construct compact fingerprints incrementally, and these fingerprints are a close approximation of page-wise similarity among VMs. For the discussion below, we only consider identical pages for computing similarity, but it is easy to extend the algorithm to take into account similarity at sub-page granularity.

Let A and B denote the list of pages (including duplicates) belonging to VM_A and VM_B respectively. We wish to measure the *similarity* S, defined as follows:

$$S = \frac{|A \cup B|}{|A| + |B|}$$

S captures the total number of distinct pages required to support the two VMs, should they be co-located. Let A' and B' denote the *set* of distinct pages belonging to above mentioned VMs. Define the quantity M as follows:

$$M = \frac{|A' \cap B'|}{|A' \cup B'|}$$

M is the quantity that the min-wise hash based signatures approximate. Specifically, we consider a family of min-wise independent hash functions $h_1, \ldots, h_k$. The signature for each VM then is simply the set of pages which when hashed with the corresponding hash function yield the minimum hash value for that function across all the pages. We can compute M in an online manner as follows. Recall that the NRU clock in DIFFERENCE ENGINE walks through pages in physical memory. For each page that the clock encounters, we update the signature for the VM that owns the page as follows. For each of the hash functions $h_1, \ldots, h_k$, if the computed hash value is less than the current minimum hash value, we record this page as the new minimum.

Thus we can compute M in an online manner. We then wish to compute S, without actually computing the set union $|A \cup B|$. Let C_A and C_B denote the fraction of distinct pages in the two VMs, $|A'|/|A|$ and $|B'|/|B|$ respectively. Further, assume that both the VMs have the same amount of allocated memory (same number of pages, N). Then we have:

$$\begin{aligned}
\frac{C_A + C_B}{2} \times \frac{1}{1+M} &= \left(\frac{|A'|}{|A|} + \frac{|B'|}{|B|}\right) \times \frac{|A' \cup B'|}{|A'| + |B'|} \\
&= \left(\frac{|A'| + |B'|}{N}\right) \times \frac{|A' \cup B'|}{|A'| + |B'|} \\
&= \frac{|A' \cup B'|}{2N} \\
&= \frac{|A \cup B|}{|A| + |B|} = S
\end{aligned}$$

The last step follows from the fact that the set union discards duplicate pages, and therefore $|A' \cup B'|$ is identical to $|A \cup B|$. Thus, knowing M, one can quickly compute S.

The algorithm then is to compute M for each VM locally, and communicate only the min-wise hash signatures to a central site. S can then be quickly computed and pair-wise comparisons can be made to determine similarity. Note that since M is a vector, clustering algorithms can also be used to quickly group similar VMs together.

9.1.2 Exploiting Improved Hardware Support

Several engineering challenges addressed in this dissertation could be simplified with improving hardware support for virtualization. For instance, the global clock in Difference Engine uses the shadow page table mechanism (in software) to track infrequently used pages. Newer hardware from both Intel and AMD implement the shadow page table functionality in hardware, thus allowing for a more efficient implementation.

Likewise, one of the benefits of Difference Engine is that it can support a given set of VMs using lesser memory. In the near future, memory hardware will provide the capability to selectively turn off idle banks. Thus, using Difference Engine one could compress the existing VMs onto a fewer memory banks and the remaining could then be shut down to save power.

The general direction here is that as hardware support for virtualization improves, we should exploit it to optimize existing features and add new ones. The next generation of I/O devices — network interface cards, hard drive controllers, PCI bus controllers and so on — will all have virtualization support built in.

9.1.3 Scaling Low-level Subsystems

Time dilation uniformly speeds up all time-based subsystems such as the CPU and disk/network bandwidth. For resource equivalence, specially with DieCast, we need fine-grained, independent knobs to control the precise resources visible to any individual VM, such that after time dilation, the perceived resource capacity matches some given target configuration.

However, the mechanisms presented in this dissertation only cover the first-order, high-level resources such as CPU and network. Under time dilation, the low-level subsystems such as the PCI bus and the memory bus will also appear faster. Conversely, we need mechanisms to control the exact behavior of these low-level resources as perceived by a VM in order to guarantee resource equivalence at this granularity.

However, this is a much more challenging problem than scaling the network or the disk, for example. Interposing on each memory request in order to artificially delay it (like we did for disk I/O) is prohibitive in software. This has not been a concern so far because most applications are resilient to such low level timing issues. The dominant latencies usually lie elsewhere in the system and our mechanisms appropriately manage those, hence the vast majority of applications are still valid targets for time dilation and DieCast.

9.2 Limitations

Virtual machines are the building block for the work in this dissertation. But one can question the necessity of virtualization itself for the ideas presented in this dissertation. In other words, do we lose anything if all the functionality is moved from the hypervisor into the operating system?

First, note virtualization is clearly indispensable in certain situations. Virtual machines are an attractive mechanism to run legacy software on newer, unsupported hardware. Similarly, virtualization is critical for supporting binary blobs of software that can no longer be updated for changing hardware. But the question we are posing is not about the utility of virtualization itself, but about the efficacy of virtualization as a vehicle for implementing time dilation, DieCast and Difference Engine.

For instance, as we have discussed earlier, it is possible to implement time dilation directly inside an operating system. Similarly, the mechanisms used by Difference Engine— such as in-memory page compression and page patching — can be implemented directly in the operating system's virtual memory subsystem. However, there are two distinct disadvantages in doing so:

1. Operating system specific implementations lead to duplication of code and effort, since each supported OS must be modified independently. It further violate transparency with

respect to time dilation, since the OS can now clearly differentiate between the real time frame and the dilated time frame. Finally, the source code of the OS might simply be unavailable for modification.

2. An OS specific implementation would lack visibility into the global state of the system. Thus, page patching mechanisms within an OS would not be able to exploit similar pages in another co-located VM.

On the other hand, an OS specific implementation can make smarter decisions since it has better knowledge of the system state. For instance, right now the NRU clock in Difference Engine makes decisions based on shadow page table accesses. But these decisions might be in conflict with the local page replacement policies within a guest OS. Similarly, maintaining transparency incurs higher cost as well — trapping and emulating the `RDTSC` instruction for time dilation or maintaining the per-page state for the operation of the NRU clock are examples of operations that could be significantly sped up if the OS could be involved. In general, OS specific modifications usually result in better performance, since virtualization will always introduce some overhead.

It is unclear whether these trade-offs can always be balanced in the general case and the question of whether the functionality should reside in the OS or the VMM still remains open. It is instructive to note that while virtual machine monitors started out as extremely thin layers of software, contemporary hypervisors are approaching operating systems in terms of the size of their code base, functionality and complexity. For instance, the Xen VMM already does some basic memory management, protection, resource allocation and scheduling. Given this trend, it is not unimaginable that in the near future, hypervisors would evolve into yet another operating system, where the resource principles are virtual machines instead of processes. This further complicates the quest for the perfect interface between the VMM and the OS. I believe that a hybrid approach offers the best promise: OS specific code should be responsible for implementing all the local functionality (such as patching pages within the same OS), with appropriate interfaces at the VMM–OS layer for communicating global information (such as cross-VM sharing).

Apart from the limitation of virtualization as a platform, time dilation (and thus DieCast) and Difference Engine have limitations of their own. We discussed some of the limitations of time dilation and DieCast in Sections 4.3 and 6.6 respectively. Broadly speaking, all the limitations of DieCast stem from the fact that we can not accurately scale all resources at arbitrarily fine-grained granularities. For instance, consider an original system with ten physical machines. On a DieCast-scaled system, these ten machines would be encapsulated as VMs running on fewer physical machines. Clearly, the disk accesses on the machines in the test harness would be completely different than the disk accesses on the machines in the original system — completely sequential I/O on the local disks in the original system might appear as random I/O to the disks in the test harness, since each disk might be multiplexed across several VMs. Consequently, the failure characteristics of the DieCast-scaled system — such as the mean time to failure (MTTF) for a single disk — will not be the same as those in the original system. Similarly, we currently do not scale low-level subsystems such as the memory bus or the PCI bus. Consequently, memory accesses in a DieCast-scaled system will appear faster than those in the original system. Nonetheless, the vast majority of applications are impervious to such low-level timing issues and are valid targets for DieCast.

Difference Engine has limitations of a very different nature. We have implemented Difference Engine entirely in the VMM which has the advantage that no guest OS modifications are required and the VMM can leverage global knowledge for making decisions. At the same time, since Difference Engine expects no OS co-operation and thus has no visibility in a guest OS's internal state, the decisions it makes could be sub-optimal. As we mentioned above, the virtualization layer introduces some non-zero overhead as well. A more interesting limitation arises from the way Difference Engine is used: presumably the memory saved by Difference Engine will be used to create additional VMs. Thus the memory saved is useful only if it can actually accommodate additional VMs. Clearly then, marginal savings in memory are less useful. That is, the saved memory can be used only if the savings are non-trivial. This is a function of the size of the initial VMs, the amount of savings delivered, as well as the size of the additional VMs to be created. It is not hard to see that there will be many cases where memory will be saved, but can not be used.

Despite these limitations, we believe that our approaches to address horizontal and vertical scalability have significant value. Addressing the above mentioned limitations remains part of future work. We next summarize the contributions of this dissertation.

9.3 Summary

We began this dissertation by describing the two dimensions of scalable multiplexing: horizontal scalability refers to the limit on the number of VMs that can be supported on a single physical machine, and vertical scalability refers to the limit on the resources visible to an individual VM. Our claim was that addressing scalability along these two dimensions not only benefits existing applications of virtualization such as server consolidation, but also enables several novel and interesting applications that were not possible earlier.

Recall that scaling a system vertically in this context requires increasing the resources available to it. However, there is a hard limit: once the underlying physical resources are exhausted, we can not make any further resources available to a VM. We took the alternate approach of increasing the perceived resource capacity instead. Time dilation essentially trades off time for other resources in the system. This seemingly trivial operation yields some very interesting results, because it allows us to increase system capacity beyond the capacity of the underlying hardware.

Consider that researchers spend a great deal of effort speculating about the impacts of various technology trends. Indeed, the systems community is frequently concerned with questions of scale: what happens to a system when bandwidth increases by X, latency by Y, CPU speed by Z, etc. One challenge to addressing such questions is the cost or availability of emerging hardware technologies. Experimenting at scale with communication or computing technologies that are either not yet available or prohibitively expensive is a significant limitation to understanding interactions of existing and emerging technologies. Similarly, testing large network services remains difficult because of their scale and complexity. Ideally, a comprehensive evaluation would require running a test system identically configured to and at the same scale as the original system — but this might not be technically or economically feasible. Such testing should enable finding performance anomalies, failure recovery problems, and configuration errors under a variety of workloads and failure conditions before triggering corresponding errors during live runs.

Time dilation enables empirical evaluation of protocols and applications at speeds and capacities not currently available from production hardware. In particular, we show that time dilation enables faithful emulation of network links several orders of magnitude greater than physically feasible on commodity hardware. The technique's power is its ability to probe future scenarios using today's hardware and network protocol stacks. We have shown that under a variety of network conditions and transport protocols, time dilation yields accurate results. Further, we are able to independently scale CPU and network bandwidth, allowing researchers to experiment with radically new balance points in computation to communication ratios of new technologies.

We leverage time dilation to device a methodology and framework called DieCast, to enable system testing to more closely match both the configuration and scale of the original system. We show how to multiplex multiple virtual machines, each configured identically to a node in the original system, across individual physical machines. We then dilate individual machine resources, including CPU cycles, network communication characteristics, and disk I/O, to provide the illusion that each VM has as much computing power as corresponding physical nodes in the original system. By trading time for resources, we enable more realistic tests involving more hosts and more complex network topologies than would otherwise be possible on the underlying hardware. While our approach does add necessary storage and multiplexing overhead, an evaluation with a range of network services, including a commercial filesystem, demonstrates our accuracy and the potential to significantly increase the scale and realism of testing network services.

Thus, our mechanisms for vertical scalability enable several useful applications. However, one of the primary limitations of time dilation is that it does not scale static resources such as main memory capacity. This limits the utility of DieCast as well, since the main memory limits the number of VMs that can be created on a single physical machine and hence affects the scaling factor that a DieCast-scaled system can use. This brings us to the issue of horizontal scalability. Note that scaling a system horizontally would enable more aggressive server consolidation, which is independently useful for many other applications.

Earlier work shows that substantial memory savings are available from harvesting identical pages across virtual machines when running homogeneous workloads. In this dissertation, we show that there are significant additional memory savings available from locating and patching similar pages and in-memory page compression. We presented the design and evaluation of Difference Engine to demonstrate the potential memory savings available from leveraging a combination of whole page sharing, page patching, and compression. Specifically, Difference Engine provides:

1. algorithms to quickly identify candidate pages for patching,
2. demand paging to support over-subscription of total assigned physical memory, and
3. a clock mechanism to identify appropriate target machine pages for sharing, patching, compression and paging.

Our performance evaluation shows that Difference Engine delivers an additional factor of 1.6–2.5 more memory savings than VMware ESX Server for a variety of workloads, with minimal performance overhead. Significantly, Difference Engine is able to save more than 50% memory even for heterogeneous workloads. We further demonstrate how this saved memory can be used for creating additional VMs in support of increased aggregate system capacity.

Overall, our techniques establish that there exist alternative, more scalable approaches to resource multiplexing in virtual machines that scale both horizontally and vertically. Together, these mechanisms have the potential to increase the utility of a given infrastructure by an order of magnitude or more. While many challenges still remain, we believe that this work establishes the importance and relevance of efficient multiplexing for realizing the potential of virtualization.

Bibliography

[1] http://en.wikipedia.org/wiki/History_of_CP/CMS.

[2] http://www.idc.com/getdoc.jsp?containerId=prUS20778407.

[3] http://newsroom.cisco.com/images/2008/features/chart_virtulization_091608.jpg.

[4] http://www.vmware.com/technology/virtual-infrastructure.html.

[5] http://www.claunia.com/qemu/.

[6] http://aws.amazon.com/ec2.

[7] BitTorrent. http://www.bittorrent.com.

[8] Credit Scheduler. http://wiki.xensource.com/xenwiki/CreditScheduler.

[9] Dow Jones Plunge Fueled by Overwhelmed Computers. http://slashdot.org/article.pl?sid=07/02/28/1416236.

[10] FreeBSD Bootloader Stops with Btx Halted in HVM Domu. http://bugzilla.xensource.com/bugzilla/show_bug.cgi?id=622.

[11] IOZone Filesystem Benchmark. http://www.iozone.org/.

[12] Linux Advanced Routing and Traffic Control. http://www.lartc.org.

[13] Linux TCP Tuning Guide. http://www-didc.lbl.gov/TCP-tuning/linux.html.

[14] Lmbench - Tools for Performance Analysis. http://www.bitmover.com/lmbench/.

[15] Net:Netem. http://www.linuxfoundation.org/en/Net:Netem.

[16] Panasas. http://www.panasas.com.

[17] Panasas ActiveScale Storage Cluster Will Provide I/O for World's Fastest Computer. http://panasas.com/press_release_111306.html.

[18] Sysbench: A System Performance Benchmark. http://sysbench.sourceforge.net/.

[19] Tcpdump/libpcap Public Repository. http://www.tcpdump.org.

[20] Teragrid. http://www.teragrid.org.

[21] The Network Simulator – Ns-2. http://www.isi.edu/nsnam/ns/.

[22] Tokyo Stock Exchange Stops Trading Amid Wave of Selling. http://www.usatoday.com/money/world/2006-01-18-tokyo-ap_x.htm.

[23] VMware Appliances. http://www.vmware.com/vmtn/appliances.

[24] Xenstore. http://wiki.xensource.com/xenwiki/XenStore.

[25] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Symposium on Operating Systems Principles*, pages 74-89, Bolton Landing, New York, October 2003.

[26] AMD. Amd64 Secure Virtual Machine Architecture Reference Manual. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33047.pdf.

[27] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles*, pages 143-156, Saint Malo, France, October 1997.

[28] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, April 2005.

[29] J. Bonwick. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer Technical Conference*, pages 87-98, Boston, Massachusetts, June 1994.

[30] A. Z. Broder. Identifying and Filtering Near-duplicate Documents. In *Proceedings of the Symposium on Combinatorial Pattern Matching*, pages 1-10, London, United Kingdom, 2000.

[31] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise Independent Permutations (extended Abstract). In *Proceedings of the ACM Symposium on Theory of Computing*, pages 327-336, Dallas, Texas, May 1998.

[32] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski. Quorum: Flexible Quality of Service for Internet Services. In *Proceedings of the Symposium on Networked System Design and Implementation*, pages 159-174, Boston, Massachusetts, May 2005.

[33] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. In *IEEE Micro*, 23(2):22-28, 2003.

[34] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 259-272, San Francisco, California, December 2004.

[35] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Symposium on Operating Systems Principles*, pages 164-177, Bolton Landing, New York, October 2003.

[36] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the Symposium on Networked System Design and Implementation*, pages 161–174, San Francisco, California, 2008.

[37] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246-261, Seattle, Washington, 2002.

[38] Y.-C. Cheng, U. Hölzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proceedings of the USENIX Annual Technical Conference*, pages 87-98, Boston, Massachusetts, June 2004.

[39] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centres. In *Proceedings of the Symposium on Operating Systems Principles*, pages 103-116, Banff, Canada, October 2001.

[40] K. C. Claffy, G. C. Polyzos, and H.-W. Braun. Application of Sampling Methodologies to Network Traffic Characterization. In *Proceedings of the SIGCOMM Conference*, pages 194-203, San Francisco, California, September 1993.

[41] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web Traffic: Evidence and Possible Causes. In *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[42] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[43] R. J. Creasy. The Origin of the Vm/370 Time-sharing System. In *IBM Journal of Research and Development*, 25(5), 1981.

[44] F. Douglis. The Compression Cache: Using on-line Compression to Extend Physical Memory. In *Proceedings of the USENIX Winter Technical Conference*, pages 519-529, San Diego, California, January 1993.

[45] F. Douglis and A. Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 113-126, San Antonio, Texas, June 2003.

[46] J. Dike. A User-mode Port of the Linux Kernel. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, USENIX Association, 2000.

[47] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay. In *Proceedings of the*

Symposium on Operating System Design and Implementation, pages 211–224, Boston, Massachusetts, December 2002.

[48] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) Scheduling: Supporting Latency-sensitive Threads in a General-purpose Schedular. In *Proceedings of the Symposium on Operating Systems Principles*, pages 261-276, Kiawah Island, South Carolina, December 1999.

[49] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based Resource Provisioning in a Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, March 2003.

[50] R. S. de Castro, A. P. d. Lago, and D. D. Silva. Adaptive Compressed Caching: Design and Implementation. In *Symposium on Computer Architecture and High Performance Computing*, 2003.

[51] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `http://www.globus.org/alliance/publications/papers/ogsa.pdf`, 2002.

[52] S. Floyd. Highspeed TCP for Large Congestion Windows. IETF Request for Comments 3649, December 2003.

[53] S. Floyd and E. Kohler. Internet Research Needs Better Models. In *SIGCOMM Computer Communications Review*, 33(1):29-34, 2003.

[54] T. M. Forum. Mpi: A Message Passing Interface. In *Proceedings of the ACM/IEEE conference on Supercomputing*, Portland, Oregon, 1993.

[55] D. Gupta, K. V. Vishwanath, and A. Vahdat. Diecast: Testing Distributed Systems with an Accurate Scale Model. University of California, San Diego, Technical Report CS2007-0910, 2007.

[56] R. Goldberg. Architectural Principles for Virtual Computer Systems. Ph.D. Thesis, Harvard University, 1972.

[57] F. E. Gillett. X86 Virtualization Adopters Hit the Tipping Point. `http://www.forrester.com/Research/Document/Excerpt/0,7211,43892,00.html`.

[58] G. R. Ganger and contributors. The DiskSim Simulation Environment. `http://www.pdl.cmu.edu/DiskSim/index.html`.

[59] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the Symposium on Networked System Design and Implementation*, pages 143–158, Boston, Massachusetts, May 2005.

[60] P. Hsieh. Hash Functions. `http://www.azillionmonkeys.com/qed/hash.html`.

[61] Intel. Intel Virtualization Technology. `http://www.intel.com/technology/computing/vptech/index.htm`.

[62] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Chapter 12, 1991.

[63] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. IETF Request for Comments 1323, May 1992.

[64] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDiff Generic Differencing and Compression Data Format. IETF Request for Comments 3284, June 2002.

[65] D. Katabi, M. Handley, and C. E. Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the SIGCOMM Conference*, pages 89-102, Pittsburgh, Pennsylvania, August 2002.

[66] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the USENIX Annual Technical Conference*, pages 59-72, Boston, Massachusetts, June 2004.

[67] T. Kelly. Scalable Tcp: Improving Performance in Highspeed Wide Area Networks. In *SIGCOMM Computer Communications Review*, 33(2):83-91, 2003.

[68] J. F. Kloster, J. Kristensen, and A. Mejlholm. On the Feasibility of Memory Sharing. Master's Thesis, Aalborg University, 2006.

[69] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing Malware with Virtual Machines. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 314–327, 2006.

[70] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, April 2005.

[71] R. Love. *Linux Kernel Development*. Novell Press, 2005.

[72] S. Low. Connectix Ram Doubler Information. `http://www.lowtek.com/maxram/rd.html`, 1996.

[73] K. P. Lawton. Bochs: A Portable Pc Emulator for Unix/x. In *Linux J.*, page 7, Specialized Systems Consultants, Inc., Seattle, WA, USA, 1996.

[74] T. V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-delay Products and Random Loss. In *IEEE/ACM Transactions on Networking*, 5(3):336-350, 1997.

[75] J. Mogul. Emergent (mis) Behavior vs. Complex Software Systems. In *Proceedings of the European Conference on Computer Systems*, pages 293–304, Leuven, Belgium, April 2006.

[76] J. MacDonald. Xdelta. `http://www.xdelta.org`.

[77] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making Scheduling "cool": Temperature-aware Workload Placement in Data Centers. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, California, April 2005.

[78] K.-T. Moeller. Virtual Machine Benchmarking. Diploma Thesis, System Architecture Group, University of Karlsruhe, Germany, 2007.

[79] U. Manber and S. Wu. Glimpse: A Tool to Search through Entire File Systems. In *Proceedings of the USENIX Winter Technical Conference*, pages 23-32, San Francisco, California, January 1994.

[80] V. Makhija, B. Herndon, P. Smith, L. Roderick, E. Zamost, and J. Anderson. Vmmark: A Scalable Benchmark for Virtualized Systems. VMware, Technical Report TR 2006-002, 2006.

[81] J. C. Mogul. TCP Offload is a Dumb Idea Whose Time Has Come. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.

[82] L. I. M., M. Derek, B. Richard, R. Timothy, B. P. T., E. David, F. Robin, and H. E. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. In *IEEE Journal of Selected Areas in Communications*, 1996.

[83] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 61-76, San Francisco, California, December 2004.

[84] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proceedings of the International Conference on Supercomputing*, Seattle, Washington, 2007.

[85] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Seattle, Washington, March 2003.

[86] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the Symposium on Operating System Design and Implementation*, Seattle, Washington, November 2006.

[87] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. Shrink: A Method for Scaleable Performance Prediction and Efficient Network Simulation. In *Proceedings of the IEEE International Conference on Computer Communications*, San Francisco, California, March 2003.

[88] G. J. Popek and R. P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. In *Proceedings of the Symposium on Operating Systems Principles*, page 121, Yorktown Heights, New York, October 1973.

[89] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. In *SIGCOMM Computer Communications Review*, 27(1):31-41, 1997.

[90] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. In *ACM Transactions on Modeling and Computer Simulation*, 7(1):78-103, 1997.

[91] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proceedings of the International Symposium on Computer Architecture*, 2006.

[92] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. In *SIGCOMM Computer Communications Review*, 33(2):65-81, 2003.

[93] G. F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, pages 5-12, Karlsruhe, Germany, 2003.

[94] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the USENIX Security Symposium*, pages 10-10, Denver, Colorado, 2000.

[95] M. Stahlman. Does Google Have a Million Servers? `http://www.gartner.com/DisplayDocument?doc_cd=149024`.

[96] B. K. Szymanski, A. Saifee, A. Sastry, Y. Liu, and K. Madnani. Genesis: A System for Large-scale Parallel Network Simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 89-96, Washington, D.C., 2002.

[97] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. In *ACM Computing Surveys*, 22(4):299–319, 1990.

[98] A. Tridgell. Emulating Netbench. `http://samba.org/ftp/tridge/dbench/`.

[99] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2007.

[100] I. C. Tuduce and T. Gross. Adaptive Main Memory Compression. In *Proceedings of the USENIX Annual Technical Conference*, pages 255–289, Anaheim, California, April 2005.

[101] B. Urgaonkar, P. J. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 239–254, Boston, Massachusetts, December 2002.

[102] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. S. Chase, and D. Becker. Scalability and Accuracy in a Large-scale Network Emulator. In *Proceedings of the Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[103] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the Symposium on Operating Systems Principles*, pages 148-162, Brighton, United Kingdom, October 2005.

[104] VMware. ESX Server 3 Configuration Guide. `http://www.vmware.com/pdf/vi3_35/esx_3/r35/vi3_35_25_3_server_config.pdf`.

[105] VMware. Timekeeping in VMware Virtual Machines. `http://www.vmware.com/pdf/vmware_timekeeping.pdf`.

[106] VMware. VMware P2V Assistant. `http://www.vmware.com/products/p2v/`.

[107] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, Santa Fe, New Mexico, May 2005.

[108] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[109] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[110] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Symposium on Operating System Design and Implementation*, Boston, Massachusetts, December 2002.

[111] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 101-116, Monterey, California, June 1999.

[112] P. R. Warkhede, S. Suri, and G. Varghese. Fast Packet Classification for Two-dimensional Conflict-free Filters. In *Proceedings of the IEEE International Conference on Computer Communications*, pages 1434-1443, Anchorage, Alaska, April 2001.

[113] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BiC) for Fast Long-distance Networks. In *Proceedings of the IEEE International Conference on Computer Communications*, Hong Kong, China, March 2004.