DieCast: Testing Distributed Systems with an Accurate Scale Model

Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat
University of California, San Diego
{dgupta,kvishwanath,vahdat}@cs.ucsd.edu

Abstract

Large-scale network services can consist of tens of thousands of machines running thousands of unique software configurations spread across hundreds of physical networks. Testing such services for complex performance problems and configuration errors remains a difficult problem. Existing testing techniques, such as simulation or running smaller instances of a service, have limitations in predicting overall service behavior.

Although technically and economically infeasible at this time, testing should ideally be performed at the same scale and with the same configuration as the deployed service. We present DieCast, an approach to scaling network services in which we multiplex all of the nodes in a given service configuration as virtual machines (VM) spread across a much smaller number of physical machines in a test harness. CPU, network, and disk are then accurately scaled to provide the illusion that each VM matches a machine from the original service in terms of both available computing resources and communication behavior to remote service nodes. We present the architecture and evaluation of a system to support such experimentation and discuss its limitations. We show that for a variety of services—including a commercial, high-performance, cluster-based file system—and resource utilization levels, DieCast matches the behavior of the original service while using a fraction of the physical resources.

1 Introduction

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 tens of thousands of machines are commonplace [11].

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 [24] 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.

One of the primary challenges facing designers of modern network services is testing their dynamically evolving system architecture. 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 [21], the effects of correlated failures [20], 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.

The goal of this work is to develop a testing methodology and architecture that can accurately predict the behavior of modern network services while employing an order of magnitude less hardware resources. For example, consider a service consisting of 10,000 heterogeneous machines, 100 switches, and hundreds of individual software configurations. We aim to configure a smaller number of machines (e.g., 100-1000 depending on service characteristics) to emulate the original configuration as closely as possible and to subject the test infrastructure to the same workload and failure conditions as 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.

A key insight behind our work is that we can trade *time* for system capacity while accurately scaling individual system components to match the behavior of the target infrastructure. We employ *time dilation* to accurately scale the capacity of individual systems by a configurable factor [19]. Time dilation fully encapsulates operating systems and applications such that the rate at which time passes can be modified by a constant factor. A time dilation factor (TDF) of 10 means that for every second of real time, all software in a dilated frame believes that time has advanced by only 100 ms. If we wish to subject a target system to a one-hour workload when scaling the system by a factor of 10, the test would take 10 hours of real time. For many testing environments, this is an appropriate tradeoff. Since the passage of time is slowed down while the rate of *external events* (such as network I/O) remains unchanged, the system appears to have substantially higher processing power and faster network and disk.

In this paper, we present DieCast, a complete environment for building accurate models of network services (Section 2). Critically, we run the actual operating systems and application software of some target environment on a fraction of the hardware in that environment. This work makes the following contributions. First, we extend our original implementation of time dilation [19] to support fully virtualized as well as paravirtualized hosts. To support complete system evaluations, our second contribution shows how to extend dilation to disk and CPU (Section 3). 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 (Sections 4 and 5). The goals of this work are ambitious and while we cannot claim to have addressed all of the myriad challenges associated with testing large-scale network services (Section 6), we believe that DieCast shows significant promise as a testing vehicle

2 System Architecture

We begin by providing an overview of our approach to scaling a system down to a target test harness. We then discuss the individual components of our architecture.

2.1 Overview

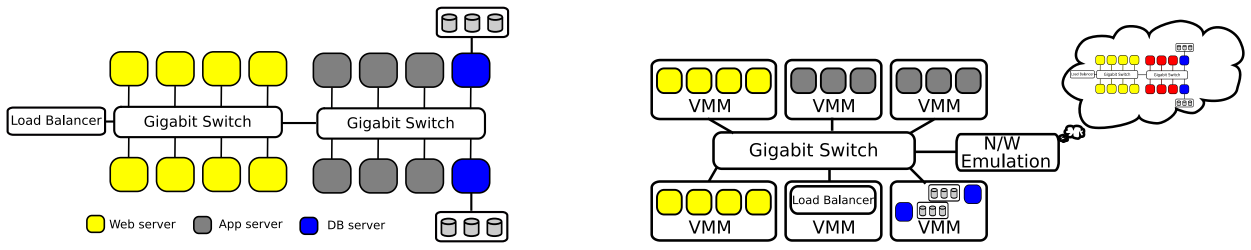
Figure 1 gives an overview of our approach. On the left (Figure 1(a)) is an abstract depiction of a network service. A load balancing switch sits in front of the service and redirects requests among a set of front-end HTTP servers. These requests may in turn travel to a middle tier of application servers, who may query a storage tier consisting of databases or network attached storage.

Figure 1(b) shows how a target service can be scaled with DieCast. We encapsulate all nodes from the original service in virtual machines and multiplex several of these VMs onto physical machines in the test harness. Critically, we employ time dilation in the VMM running on each physical machine to provide the illusion that each virtual machine has, for example, as much processing power, disk I/O, and network bandwidth as the corresponding host in the original configuration despite the fact that it is sharing underlying resources with other VMs. DieCast configures VMs to communicate through a network emulator to reproduce the characteristics of the original system topology. We then initialize the test system using the setup routines of the original system and subject it to appropriate workloads and fault-loads to evaluate system behavior.

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.

2.2 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



(a) Original System (b) Test System

Figure 1: Scaling a network service to the DieCast infrastructure.

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 the characteristics of the target system. Section 6 highlights the potential limitations of DieCast scaling. In general, scaling accuracy will degrade with: i) application sensitivity to the fine-grained timing behavior of external hardware devices; ii) capacity-constrained physical resources; and iii) 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 mispredict 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 Sections 4 and 5.

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. For example, one partial solution is to configure the test system with more memory and storage than the original system. While this 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.

Looking forward, ongoing research in VMM architectures have the potential to reclaim some of the memory [32] and storage overhead [33] 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.

2.3 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 work, 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 [7], 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.

2.4 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 [26]. 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.

DieCast then configures the VMs such that each VM appears to have resources identical to a physical machine in the original system. Consider a physical machine hosting 10 VMs. DieCast would run each VM with a scaling factor of 10, but allocate each VM only 10% of the actual physical resource. 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 in the VMs time frame, resources appear equivalent to the original machine. We only explicitly scale CPU and disk I/O latency on the host; scaling of network I/O happens via network emulation as described next.

2.5 Network Emulation

The final 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 [27], Modelnet [31] and Netem [3] where appropriate.

It is likely that the bisection bandwidth of the original service topology will be larger than that available in the test system. Fortunately, time dilation is of significant value here. Convincing a virtual machine scaled by a factor of 10 that it is receiving data at 1 Gbps only requires forwarding data to it at 100 Mbps. Similarly, it may appear that latencies in an original cluster-based service may be low enough that the additional software forwarding overhead associated with the emulation environment could make it difficult to match the latencies in the original network. 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.

Note that the scaling factor need not match the TDF. For example, if the original network topology is so large/fast that even with a TDF of 10 the network emulator is unable to keep up, it is possible to employ a time dilation factor of 20 while maintaining a scaling factor of 10. In such a scenario, there would still on average be 10 virtual machines multiplexed onto each physical machine, however the VMM scheduler would allocate only 5% of the physical machine's resources to individual machines (meaning that 50% of CPU resources will go idle). The TDF of 20, however, would deliver additional capacity to the network emulation infrastructure to match the characteristics of the original system.

2.6 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 [15] shows how to replay a measured TCP request stream sent to a large-scale network service. For this work, 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 Implementation

We have implemented DieCast support on several versions of Xen [10]: v2.0.7, v3.0.4, and v3.1 (both paravirtualized and fully virtualized VMs). Here we focus on the Xen 3.1 implementation. We begin with a brief overview of time dilation [19] and then describe the new features required to support DieCast.

3.1 Time Dilation

Critical to time dilation is a VMM's ability to modify the perception of time within a guest OS. Fortunately, most

VMMs already have 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. Since the guest OS running in a VM does not run continuously, VMMs periodically synchronize the guest OS time with the physical machine's clock. The only requirement for a VMM to support time dilation is this ability to modify the VM's perception of time. In fact, as we demonstrate in Section 5, the concept of time dilation can be ported to other (non-virtualized) environments.

Operating systems employ a variety of time sources to keep track of time, including timer interrupts (e.g., the Programmable Interrupt Timer or PIT), specialized counters (e.g., the TSC on Intel platforms) and external time sources such as NTP. Time dilation works by intercepting the various time sources and scaling them appropriately to fully encapsulate the OS in its own time frame.

Our original modifications to Xen for paravirtualized hosts [19] therefore appropriately scale time values exposed to the VM by the hypervisor. Xen exposes two notions of time to VMs. Real time is the number of nanoseconds since boot, and wall clock time is the traditional Unix time since epoch. 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. Dilation uses a per-domain TDF variable to appropriately scale real time and wall clock time. It also scales the frequency of timer interrupts delivered to a guest OS since these timer interrupts often drive the internal time keeping of a guest. Given these modifications to Xen, our earlier work showed that network dilation matches undilated baselines for complex per-flow TCP behavior in a variety of scenarios [19].

3.2 Support for OS diversity

Our original time dilation implementation only worked with paravirtualized machines, with two major drawbacks: it supported only Linux as the guest OS, and the guest kernel required modifications. Generalizing to other platforms would have required code modifications to the respective OS. To be widely applicable, DieCast must support a variety of operating systems.

To address these limitations, we ported time dilation to support *fully virtualized* (FV) VMs, enabling DieCast to support unmodified OS images. Note that FV VMs require platforms with hardware support for virtualization, such as Intel VT or AMD SVM. While Xen support for fully virtualized VMs differs significantly from the paravirtualized VM support in several key areas such as I/O emulation, access to hardware registers, and time management, the general idea behind the implementation remains the same: we want to intercept all sources of time and scale them.

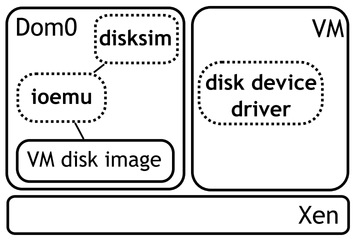
In particular, our implementation scales the PIT, the TSC register (on x86), the RTC (Real Time Clock), the ACPI power management timer and the High Performance Event Timer (HPET). As in the original implementation, we also scale the number of timer interrupts delivered to a fully virtualized guest. We allow each VM to run with an independent scaling factor. Note, however, that the scaling factor is fixed for the life time of a VM—it can not be changed at run time.

3.3 Scaling Disk I/O and CPU

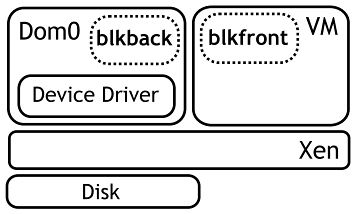
Time dilation as described in [19] did not scale disk performance, making it unsuitable for services that perform significant disk I/O. Ideally, we would scale individual disk requests at the disk controller layer. The complexity of modern drive architectures, particularly the fact that much low level 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 paravirtualized (PV) VMs and one for fully virtualized (FV) VMs. DieCast provides mechanisms to scale disk I/O for both models.

For FV VMs, DieCast integrates a highly accurate and efficient disk system simulator — Disksim [17] — which gives us a good trade-off between realism and accuracy. Figure 2(a) 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.

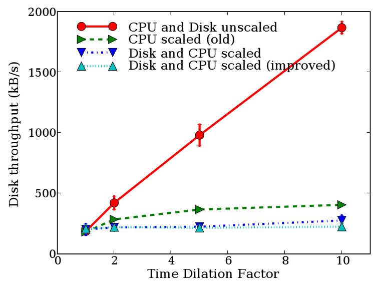
After servicing each request (but before returning), `ioemu` forwards the request to Disksim, which then returns the time, rt, the request would have taken in its simulated disk. Since we are effectively layering a software disk on top of `ioemu`, each request should ideally take exactly time rt in the 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 $(tdf * rt) - dt - st$.



(a) I/O Model for FV VMs



(b) I/O Model for PV VMs



(c) DBench throughput under Disksim

Figure 2: Scaling Disk I/O

The architecture of Disksim, however, is not amenable to integration with the PV I/O model (Figure 2(b)). 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 PV 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 PV 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 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 of disk scaling for PV 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. For brevity, we only describe one of them here. We run DBench [29] — a popular hard-drive and file-system benchmark — under different dilation factors and plot the reported throughput. Figure 2(c) shows the results for the FV I/O model with Disksim integration (results for the PV implementation can be found in a separate technical report [18]). 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% 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. Recall that we scale the CPU by bounding the amount of CPU available to each VM. Initially, we simply used Xen's Credit scheduler to allocate an appropriate fraction of CPU resources to each VM in non-work conserving mode. However, simply scaling the CPU does not govern how those CPU cycles are distributed across time. With the original Credit scheduler, if a VM does not consume its full timeslice, it can be scheduled again in subsequent timeslices. For instance, if a VM is set to be dilated by a factor of 10 and if it consumes less than 10% of the CPU in each time slice, then it will run in *every* time slice, since in aggregate it never consumes more than its hard bound of 10% of the CPU. This potential to run continuously distorts the performance of I/O-bound applications under dilation, and in particular they'll have a different timing distribution than they would in the real time frame. This distortion increases with increasing TDF. Thus, we found that, for some workloads, we may actually wish to enforce that the VM's CPU consumption should be more *uniformly* enforced across time.

We modified 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 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. The final line in figure 2(c) 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.

4 Evaluation

We seek to answer the following questions with respect to DieCast-scaling: i) 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? ii) How well does the performance of a scaled service running on fewer resources match the performance of a baseline service running with more resources? we consider three different systems: i) BitTorrent, a popular peer-to-peer file sharing program; ii) RUBiS, an auction service prototyped after eBay; and iii) Isaac, our configurable network three-tier service that allows us to generate a range of workload scenarios.

4.1 Methodology

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 evaluated DieCast implementations for several versions of Xen, we only present the results for the Xen 3.1 implementation here. Detailed evaluation for Xen 3.0.4 can be found in our technical report [18].

Each physical machine in our testbed is a dual-core 2.3GHz Intel Xeon with 4GB 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 paravirtualized setup, we do have evaluation with physical machines as the baseline. We refer the reader to [18] 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 3.3.

We configure each virtual machine with 256MB 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 characeteristics 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 3.3.

4.2 BitTorrent

We begin by using DieCast to evaluate BitTorrent [1] — a popular P2P application. For our baseline experiments, we run BitTorrent (version 3.4.2) on a total of 40 virtual machines. We configure the machines to communicate across a ModelNet-emulated dumbbell topology (Figure 3), 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 10Mbps dumbbell link and constrained access links of 10Mbps. All links have a one-way latency of 5ms. We run a total of 40 clients (with half on each side of the dumbbell). Figure 5 plots the cumulative distribution of transfer times across all clients for different file sizes (10MB and 50MB). We show the baseline 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 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 5. 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 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 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.

Finally, we consider an experiment that demonstrates the flexibility of DieCast to reproduce system performance under a variety of resource configurations starting with the same baseline. Figure 7 shows that in addition to matching 1:10 scaling using 4 physical machines hosting 10 VMs each, we can also match an alternate configuration of 8 physical machines, hosting five VMs each with a dilation factor of five. This demonstrates that even if it is necessary to vary the number of physical machines available for testing, it may still be possible to find

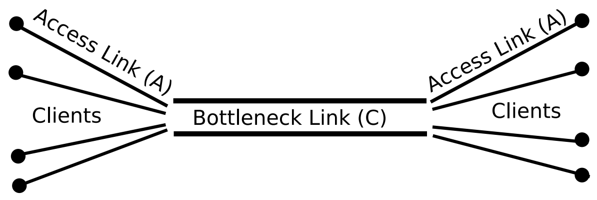


Figure 3: Topology for BitTorrent experiments.

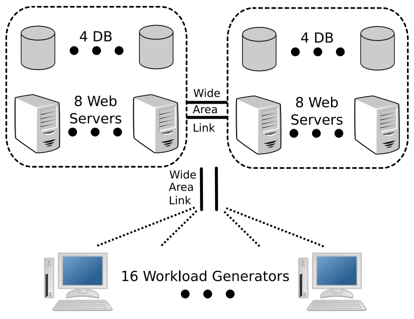


Figure 4: RUBiS Setup.

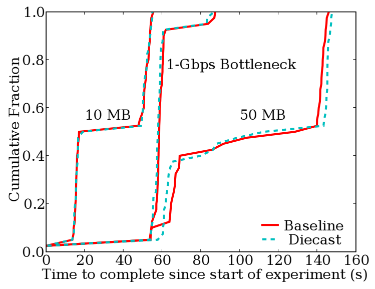


Figure 5: Performance with varying file sizes.

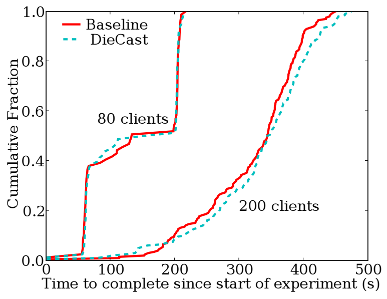


Figure 6: Varying #clients.

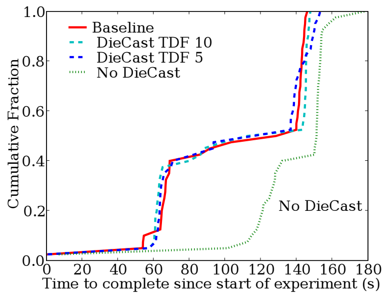


Figure 7: Different configurations.

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 nave approach varies widely from actual behavior.

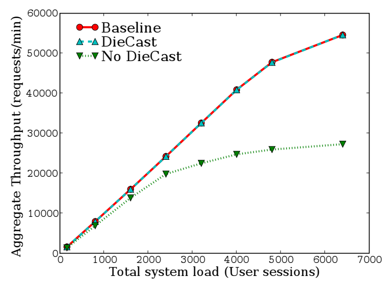
4.3 RUBiS

Next, we investigate DieCast's ability to scale a fully functional Internet service. We use RUBiS [6]—an auction site prototype designed to evaluate scalability and application server performance. RUBiS has been used by other researchers to approximate realistic Internet Services [12–14].

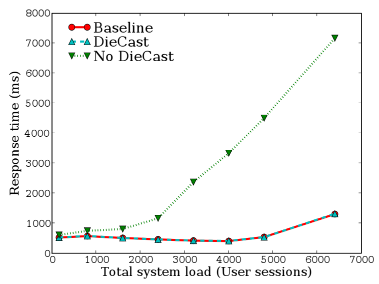
We use the PHP implementation of RUBiS running Apache as the web server and MySQL as the database. For consistent results, we re-create the database and pre-populate it with 100,000 users and items before each experiment. We use the default read-write transaction table for the workload that exercises 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 4. A 100 Mbps network link connects 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. 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 8(a) and 8(b) 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: the workload had commenting-related operations enabled, but the relevant tables were missing from the database. This led to an approximately 25% error rate



(a) Throughput



(b) Response Time

Figure 8: Comparing RUBiS application performance: Baseline vs. DieCast.

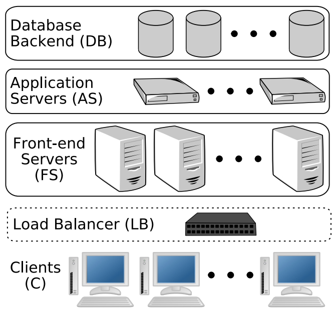
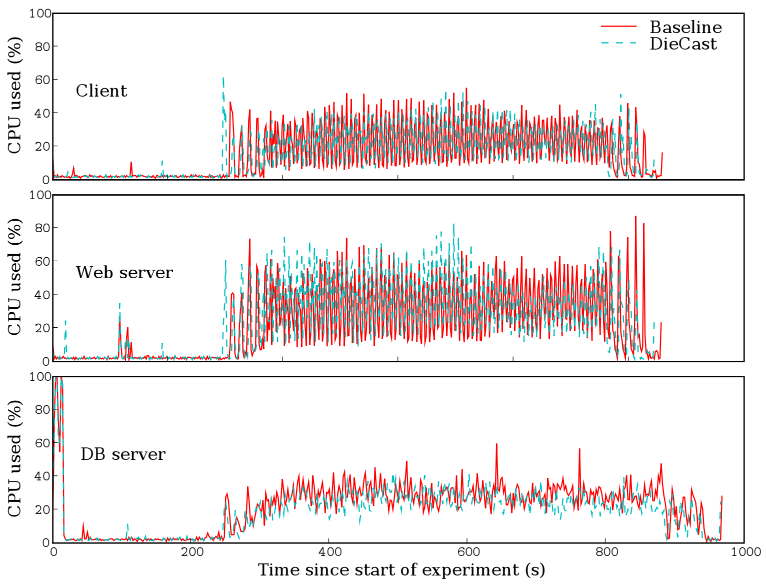
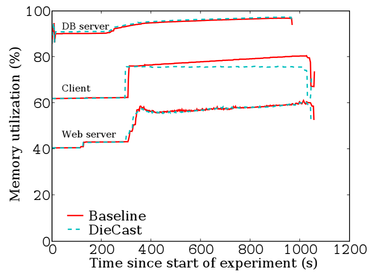


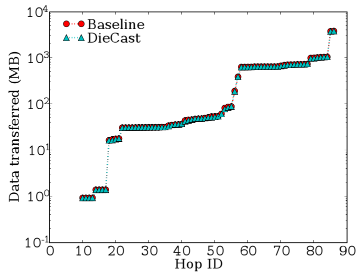
Figure 10: Architecture of Isaac.



(a) CPU profile



(b) Memory profile



(c) Network profile

Figure 9: Comparing resource utilization for RUBiS: DieCast can accurately emulate the baseline system behavior.

with similar timings in the responses to clients in both the baseline and DieCast configurations. These types of configuration errors are one example of the types of testing that we wish to enable with DieCast.

Next, Figures 9(a) and 9(b) 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 (DB 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 9(c) 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 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.

4.4 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 10 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

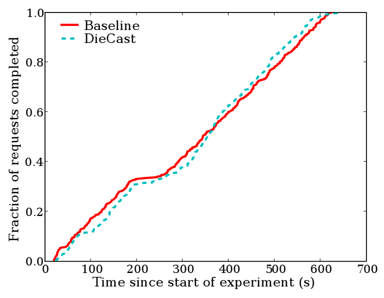


Figure 11: Request completion time.

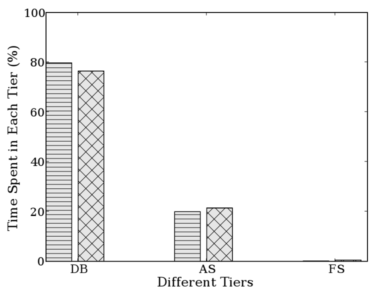


Figure 12: Tier-breakdown.

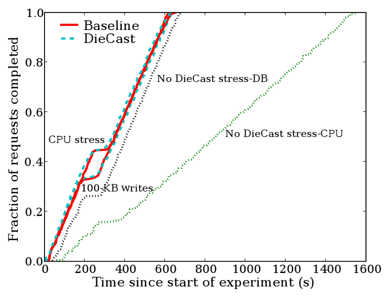


Figure 13: Stressing DB/CPU.

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.

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.

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 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. For brevity, we do not show the results of initial tests validating DieCast accuracy (in all cases, performance matched closely in both the dilated and baseline case). 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 11 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 12 shows that in addition to the end-to-end response time, DieCast closely tracks the system behavior on a per-tier basis.

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. First, we evaluate DieCast's ability to scale network services when database access dominates per-request service time. Figure 13 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. This amounts to a total of 1 MB of database writes for every request from a client. 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 2(c) 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 too are shown in Figure 13. 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 stress-DB and stress-CPU configurations with 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.

5 Commercial System Evaluation

While we were encouraged by DieCast's accuracy for the applications we considered in Section 4, all of the experiments were designed by DieCast authors and were largely academic in nature. To understand the generality of our system, we consider its applicability to a large-scale commercial system.

Panasas [4] builds scalable storage systems targeting Linux cluster computing environments. It has supplied solutions to several government agencies, 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 [5]. 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

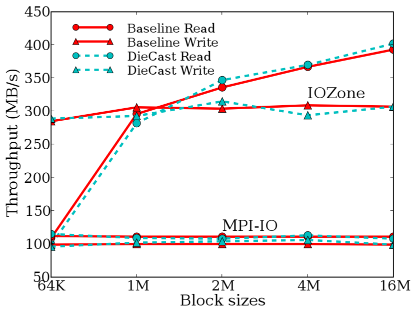


Figure 14: Validating DieCast on PanFS.

which was unavailable in the hardware Panasas was using. Even if we had the hardware, Xen did not support FreeBSD on FV VMs until recently due to a well known bug [2]. 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.

To implement time dilation in the PanFS kernel, we scale the various time sources , 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.

For scaling the network, we use Dummynet [27], 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 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.

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.

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 an PanFS server with 20 disks of capacity 250GB each (5TB 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 [23]. The second benchmark uses the Message Passing Interface (MPI) across 100 clients (again, on separate machines) [28].

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 14 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 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.

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

Table 1: Aggregate read/write throughputs from the IOZone benchmark with block size 16M. PanFS performance scales gracefully with larger client populations.

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 16MB 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 1 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 leave a performance evaluation when removing this bottleneck to future work.

We would like to emphasize that prior to our experiment, Panasas had been unable to perform experiments at this scale. This 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 DieCast Usage Scenarios

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).

It 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 faultloads 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 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 [22,24] are also known to account for a significant fraction of service failures. 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.

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 paper. 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 paper, 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 becomes 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.

7 Related Work

Our work builds upon previous efforts in a number of areas. We discuss each in turn below.

Testing scaled systems SHRiNK [25] 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 100Mbps flows through a 1Gbps pipe should be similar to 10Mbps through a 100Mbps 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.

Testing through Simulation and Emulation 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 to understanding coarse-grained performance characteristics of certain configurations. Simulation is less suited to configuration errors or to capturing the effects of unexpected component interactions, failures, etc.

Superficially, emulation techniques (e.g. Emulab [34] or ModelNet [31]), offer a more realistic alternative to simulation because they support running unmodified applications and operating systems. Unfortunately, such emulation is 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. For instance, multiplexing 1000 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 1000 machines in a multi-tier network service consisting of dozens of racks and high-speed switches.

Time Dilation DieCast leverages earlier work on Time Dilation [19] to assist with scaling the network configuration of a target service. This earlier work focused on evaluating network protocols on next-generation networking topologies, e.g., the behavior on TCP on 10Gbps Ethernet while running on 1Gbps Ethernet. Relative to this previous work, DieCast improves upon time dilation to scale *down* a particular network configuration. In addition, we demonstrate that it is possible to trade time for compute resources while accurately scaling CPU cycles, complex network topologies, and disk I/O. Finally, we demonstrate the efficacy of our approach end-to-end for complex, multi-tier network services.

Detecting Performance Anomalies There have been a number of recent efforts to debug performance anomalies in network services, including Pinpoint [14], MagPie [9], and Project 5 [8]. 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 our work is to test particular software configurations at scale to locate errors before they affect a live service.

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 [16, 30] or to reroute requests in the face of component failures [12]. 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.

8 Conclusion

Testing network services remains difficult because of their scale and complexity. While not technically or economically feasible, a comprehensive evaluation would require running a test system identically configured to and at the same scale as the original system. 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.

In this paper, we present a methodology and framework 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.

Acknowledgements

The authors would like to thank Tejasvi Aswathanarayana, Jeff Butler and Garth Gibson at Panasas for their guidance and support in porting DieCast to their systems. We would also like to thank Marvin McNett and Chris Edwards for their help in managing some of the infrastructure. Finally, we would like to thank our shepherd Steve Gribble, and our anonymous reviewers for their time and insightful comments—they helped tremendously in improving the paper.

References

[1] BitTorrent. `http://www.bittorrent.com`.

[2] FreeBSD bootloader stops with BTX halted in hvm domU. `http://bugzilla.xensource.com/bugzilla/show_bug.cgi?id=622`.

[3] Netem. `http://linux-net.osdl.org/index.php/Netem`.

[4] Panasas. `http://www.panasas.com`.

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

[6] RUBiS. `http://rubis.objectweb.org`.

[7] Vmware appliances. `http://www.vmware.com/vmtn/appliances/`.

[8] 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 19th ACM Symposium on Operating System Principles*, 2003.

[9] P. Barham, A. Doelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.

[11] L. A. Barroso, J. Dean, and U. Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 2003.

[12] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski. Quorum: Flexible Quality of Service for Internet Services. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2002.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks*, 2002.

[15] Y.-C. Cheng, U. Hoelzle, 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*, 2004.

[16] R. Doyle, J. Chase, O. Asad, W. Jen, and A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 2003.

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

[18] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Network Services with an Accurate 1/10 Scale Model. Technical Report CS2007-0910, University of California, San Diego, 2007.

[19] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, G. M. Voelker, and A. Vahdat. To Infinity and Beyond: Time-Warped Network Emulation. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006.

[20] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2005.

[21] J. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. In *Proceedings of the first EuroSys Conference*, 2006.

[22] 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 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[23] W. Norcott and D. Capps. IOzone Filesystem Benchmark. `http://www.iozone.org/`.

[24] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[25] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. SHRINK: A Method for Scalable Performance Prediction and Efficient Network Simulation. In *IEEE INFOCOM*, 2003.

[26] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. In *SIGCOMM Computer Counications Review*, volume 33, 2003.

[27] L. Rizzo. Dummynet and Forward Error Correction. In *Proceedings of the USENIX Annual Technical Conference*, 1998.

[28] The MPI Forum. MPI: A Message Passing Interface. pages 878–883, Nov. 1993.

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

[30] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[31] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[32] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[33] A. Warfield, R. Ross, K. Fraser, C. Limpach, and H. Steven. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*.

[34] 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 5th USENIX Symposium on Operating Systems Design and Implementation*, 2002.