

# Twinkle: A Fast Resource Provisioning Mechanism for Internet Services

Jun Zhu, Zhefu Jiang, Zhen Xiao  
School of Electronics Engineering and Computer Science  
Peking University, Beijing, China 100871  
{zhujun, jzf}@net.pku.edu.cn, xiaozhen@pku.edu.cn

**Abstract**—A key benefit of Amazon EC2-style cloud computing service is the ability to instantiate a large number of virtual machines (VMs) on the fly during flash crowd events. Most existing research focuses on the policy decision such as when and where to start a VM for an application. In this paper, we study a different problem: how can the VMs and the applications inside be brought up as quickly as possible? This problem has not been solved satisfactorily in existing cloud services.

We develop a fast start technique for cloud applications by restoring previously created VM snapshots of fully initialized application. We propose a set of optimizations, including working set estimation, demand prediction, and free page avoidance, that allow an application to start running with only partially loaded memory, yet without noticeable performance penalty during its subsequent execution. We implement our system, called Twinkle, in the Xen hypervisor and employ the two-dimensional page walks supported by the latest virtualization technology. We use the RUBiS and TPC-W benchmarks to evaluate its performance under flash crowd and failure over scenarios. The results indicate that Twinkle can provision VMs and restore the QoS significantly faster than the current approaches.

## I. INTRODUCTION

Internet services can experience sudden, unexpected surge in demand, or the so-called flash crowd events [1]. Those services are increasingly deployed in the cloud to take advantage of its auto scaling feature [2]. A well-known example of the cloud model is the Amazon EC2 service which allows users to rent VM instances and operate them much like raw hardware. A key benefit of using such a service is the ability to provision a large number of VM instances when flash crowd happens. Most existing work on auto scaling focuses on policy decision such as when and where to provision a VM for an Internet service. In this paper, we study the mechanism of resource provisioning: how can the VMs and the applications inside be brought up and start running as quickly as possible? This is important to the user experience, especially during flash crowd and failure over scenarios.

Unfortunately, the startup latency for cloud applications can be quite high. For example, it can take several minutes to acquire a new VM instance in Amazon EC2 [3], then comes the also time-consuming process of VM startup and application initialization. Complex applications can take a long time to initialize. This does not work well in flash crowd scenario. For example, on September 11th of 2001, the demand on the CNN’s web site increased by an order of magnitude in 15 minutes [4]. If additional capacity cannot be

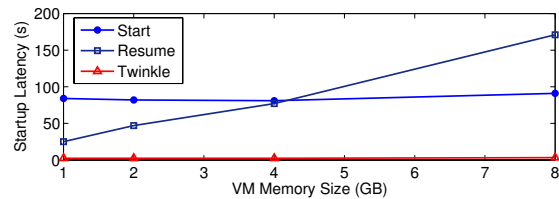


Fig. 1. Comparison of the startup latency

provisioned fast enough, a significant number of user requests will experience poor QoS or the service may even be brought down. Another common scenario is when failures occur and the application replicas need to be restarted. Optimizing the startup latency can reduce the failure over time and improve the availability of the system.

In this paper, we show how to use virtualization technology to reduce the application start time. Our technique is implemented in the Xen hypervisor [5] and is transparent to the guest operating system (guest OS) and the application. The current VM *suspend* operation allows us to take a snapshot of the application together with its running environment. The snapshot can then be resumed to restore the application later to bypass the often lengthy application initialization. Fig 1 compares this technique with the normal startup process for a typical Internet application, a RUBiS [6] service within JBoss, running on Dell PowerEdge blade servers with Intel 5620 CPU, 24 GB RAM and 10K RPM SAS disks. The VMs are configured with a varying amount of memory. The ‘Start’ curve shows the normal start time for the VM and the application. It is close to 100 seconds, independent of the memory size. We consider the application as fully started when it can respond to a user request successfully. The ‘Resume’ curve shows the start time using a previously created VM snapshot. The figure indicates that VM resumption time increases with memory size due to the time to load the VM snapshot. For a 4G VM, it is almost as slow as the normal startup time. Amazon EC2 allows high throughput VM instances with tens of Gigabytes of memory [3] where this native approach is clearly infeasible.

**Contributions.** In this paper, we present Twinkle, a fast resource provisioning mechanism that reduces VM startup latency to a few seconds, as shown in Fig 1. With this improved VM startup latency, decisions of auto scaling policies can be

put into effect more readily, which can improve user experience when combating flash crowds or failure overs. The basic design of Twinkle is to load pages on demand and we propose three techniques to improve the startup time and runtime performance: *working set estimation*, *demand prediction*, and *free page avoidance*. The first technique selectively loads only the memory pages that are likely to be accessed in the near future. This allows the VM to start running with only partial memory loaded from its snapshot. The second technique loads some of the remaining memory pages in parallel with the VM execution. It uses a demand prediction algorithm to reduce the runtime overhead due to page faults. The last technique avoids loading unused memory pages which can instead be allocated directly by the hypervisor.

We use copy-on-write (COW) technique to eliminate the need of copying root file system when a large number of VM instances are started from the same snapshot. The snapshot of the original root file system is immutable. Each new VM modifies and stores its own state locally. In addition, we propose a network reconfiguration mechanism to adapt a VM’s network after its startup.

We evaluate the performance of our system under flash crowd and failure over scenarios using Spec CPU2006, RUBiS, and TPC-W benchmarks. The results indicate that Twinkle can provision the virtual machines and restore the QoS significantly faster than the current approaches.

The remainder of this paper is organized as follows. Section II introduces the two-dimensional page walks in a virtualized system. Section III illustrates the architecture of Twinkle. Section IV and V describe the design and implementation of the key techniques that enable fast resource provisioning. Section VI evaluates the performance and applications of our system. Section VII discusses the related work, and Section VIII concludes this paper.

## II. TWO-DIMENSIONAL PAGE WALKS

Twinkle relies on the mechanism of two-dimensional page walks which is recently present in AMD’s “Barcelona” and Intel’s “Nehalem” families respectively [7][8]. Before we present the design details, in this section, we first have a brief look at the features of this hardware.

In a virtualized platform, multiple guest OSs run simultaneously, sharing the common memory with strong isolation. The hypervisor provides each guest OS with an illusion that it has a continuously zero-based physical address space, which is commonly achieved by two-dimensional page walks, as illustrated in Fig 2. There is one page walker for each dimension. The first page walker performs translation from the *virtual address* of a program to a *physical address* of the VM, and the second page walker performs translation from the physical address to a *machine address* of the physical server.

In the first generation of virtualization, the second dimensional page walk was maintained by a software-based technique, called shadow paging. Recently, to avoid software overhead, both AMD and Intel added *Nested Paging* (called *Extended Paging* by Intel) as the second hardware page walker.

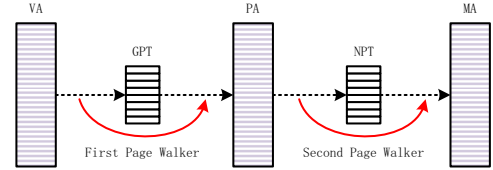


Fig. 2. Two-Dimensional Page Walks. VA stands for the virtual address of a program, PA stands for the physical address of a VM, and MA stands for the machine address of the physical server.

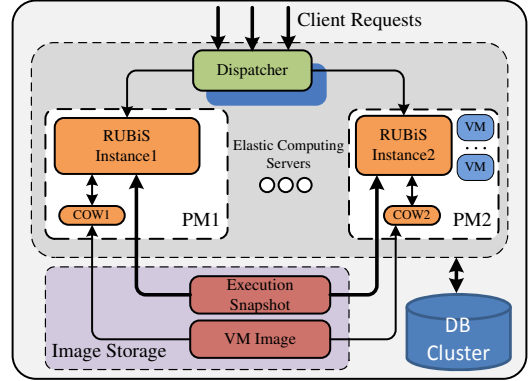


Fig. 3. Example of a cloud data center running Twinkle.

Nested Paging translates a physical address to a machine address by walking the nested page tables (NPT) that are maintained by the hypervisor. The structure of the NPT is the same as the traditional guest page table (GPT), both constructed in a hierarchical tree. Each nested page table entry contains control bits to indicate the corresponding page’s properties (e.g., *non-present*, *writable*, *readable* and *non-executable*). Twinkle exploits this hardware feature to track and manipulate VM memory layout transparently.

## III. ARCHITECTURE

The Internet services running in the clouds always employ the multi-tier architecture that consists of a presentation tier, an application logic tier and a storage tier. The first two tiers typically use non-persistent data and are often hosted in the elastic computing servers (e.g., EC2), and the persistent data is often stored in a back-end database or a distributed storage system (e.g., S3 or EBS). Different applications present burden on different tiers. The back-end storage system has addressed the scalability and reliability well, and has been successfully deployed in the clouds for years [9][10]. In this paper, we focus on the mechanism of provisioning resources in the scope of elastic computing servers.

Fig 3 shows an example of a cloud data center. The example has two physical machines (PM1 and PM2 in “Elastic Computing Servers”), each of which runs Twinkle to enable fast VM startup. The RUBiS service has two instances, each of which is encapsulated in a separate VM. If one of them fails or the workload roars, one or more instances will be provisioned on demand. Unlike existing VM startup approach, Twinkle starts a VM from a VM snapshot which contains an

initialized execution environment (“Execution Snapshot”) and a root file system (“VM Image”). The VM snapshot is stored in a shared storage (“Image Storage”), the same as EC2 which stores virtual machine images (VMI) in S3 or EBS [3].

In the front, the dispatcher monitors each VM instance and the service quality, decides how many VMs each service should be provisioned, and routes incoming requests among the VMs. In the back-end, we deploy a database cluster to ensure against bottleneck in the data tier.

#### IV. DESIGN

The objective of Twinkle is to quickly start a VM from a VM snapshot with trivial performance overhead. In this section, we will first describe the enabling techniques, *working set estimation*, *demand prediction* and *free page avoidance*. These techniques do not require any modifications to the upper operating system and the applications running in it. Then we will present the related issues of the root file system and the network reconfiguration.

##### A. Working Set Estimation

When starting from a VM snapshot, the most influential overhead on startup latency is the time of fetching the VM’s memory pages. The current approach, which starts the VM after loading its total pages, achieves VM startup in time proportional to the amount of the memory. An alternative approach is *demand paging*, which fetches the memory pages only when needed. However, loading every page from the remote storage node (Image Storage in Fig 3) is a slow process. This approach would cause the Internet services unresponsive at the beginning. The other possibility is to flush non-essential pages out of the VM’s memory with a *balloon driver* [11] to create a smaller snapshot. After the VM starts, it pages in the non-essential pages when needed. Unfortunately, balloon driver is an intrusive approach which would alter the behavior of the guest OS, and moreover, it would incur swapping or process termination if too many pages are flushed [5].

Our approach enhances startup latency by loading working set of the operating system before starting the VM. To track the working set, we borrow the idea of *post-checkpoint* tracking mechanism [12]. After the snapshot has been taken, the VM continues to run for an interval during which the pages referred by the operating system are identified as working set. This approach employs the truth that the pages referred after the snapshot will be in high probability accessed when the VM starts from the snapshot later.

In our system, there are two questions related to this post-checkpoint tracking mechanism. First, how to track the working set? In order not to intrude the guest OS, we exploit the second dimension page walker to track working set. When the snapshot has been made, we traverse the NPT of the VM, and mark all the page table entries as “*non-present*”. During the tracking period, the pages referred by the guest OS will induce a nested page fault which will be caught by the hypervisor. The pages inducing page faults are recognized as working set of the guest OS. This tracking mechanism

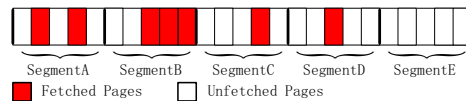


Fig. 4. Demand prediction.

is completely implemented in the hypervisor, and exerts no influence on the behavior of the guest OS.

Second, how does one determine when to stop the tracking? A large tracking window produces large working set and increases the latency of startup, while a small window reduces the size of working set but induces a number of page faults at the beginning of the recovery. In our design, we adopt an iterative and heuristic method. The post-checkpoint is divided into fixed intervals (one second each). If the number of referred pages is below ten in 60 successive iterations, the tracking is stopped.

After tracking, the working set is compressed with *gzip* and stored in the Image Storage with the VM snapshot. When starting a VM from this snapshot, the compressed working set is fetched and decompressed before starting the VM. It should be noted that we compress the working set because we load these pages as a whole before starting the VM and sequential decompression is really fast. The remaining pages in the VM snapshot are not compressed since they will be fetched on demand, and compressed snapshot cannot support random access efficiently. Also we do not compress the whole memory which was adopted by the Collective project [13], because for a VM snapshot as large as several gigabytes, the compressed snapshot may still be very large and the process of its decompression can be time-consuming.

##### B. Demand Prediction

Loading the smaller working set before VM startup is to start the VM as fast as possible. However, when attempting to refer a non-present page, the guest OS will cause a trap (called *demand-page-trap*) into the hypervisor, waiting for the hypervisor to fetch the demanded page. Handling of such demand-page-traps will incur non-trivial overhead after the VM startup. To alleviate such overhead, we introduce a demand prediction approach which fetches the pages that are most possibly needed in parallel with the VM execution.

Our approach follows the principle of memory access locality. Initially, we divide the whole memory address of the newly started VM into segments, each of which contains  $N$  consecutive pages. On each demand-page-trap, the hypervisor fetches the page and records which segment it belongs to. For one segment, if the number of demand-page-traps exceeds  $P$  pages, we fetch the remaining pages in advance due to memory access locality. Fig 4 is an example where  $N = 5$ ,  $P = 3$ . According to our policy, the remaining two pages of segment B will be fetched in advance at background, so later accesses to these pages will not result in demand-page-traps any more. In practice, the segment with the largest number of pages already fetched will be considered first.

The parameters  $N$  and  $P$  control the behavior of demand prediction. The value of  $P$  controls how close the loaded pages are to the real working set of the guest OS, while the value of  $N$  controls how our approach utilizes the memory access locality of the guest OS. In our current implementation,  $N$  and  $P$  are set as 1024 pages(4MB) and 30%. We find these values work well in the experiment presented in this paper.

### C. Free Page Avoidance

Usually, an operating system does not utilize all the memory pages, and those free pages are initially zero content for the consideration of security in a virtualized environment. For a typical Internet service, the majority of memory consumption results from dealing with client requests, while the memory requirement of the service itself is small. Based on this observation, we propose free page avoidance to further promote the performance of handling demand-page-traps when the guest OS attempts to access free but non-present pages.

When preparing the VM snapshot, the pages with zero content are identified and omitted in the snapshot. Later, during the startup phase, Twinkle checks each demanded page. If the operating system attempts to access a free page, Twinkle allocates a page to the VM and clears its content, without initiating slow I/O transmission.

### D. File System Snapshot

In our design, each Internet service holds one prepared VM snapshot in the Image Storage as shown in Fig 3. A VM snapshot has two components, an execution snapshot and a root file system image. To share the VM snapshot among all the VM instances of a service, also save disk space (as well as expenditure in the cloud), any modifications to the root file system are saved on the local storage using copy-on-write (COW) technology. The local state can be discarded when the VM instances shut down later. Currently, we use QCOW, one of the disk image formats supported by the QEMU device model (qemu-dm) integrated in Xen [14]. We create a separate COW image for each new VM. The COW image looks like a standalone image to each VM but most of its data is kept in the original image. When the VM accesses data that is not present in the COW image, the data will be fetched from the original image which the COW image is based on.

The overhead of COW image is less of an issue in our system. First, the application data is stored separately. For example, in the EC2 platform, the application data is stored in S3. In our prototype, as shown in Fig 3, we deploy a database cluster as the Internet service's data tier. In addition, disk activity is not so frequent in a modern operating system because of its highly effective cache mechanisms. Should the performance of COW image become an issue, we can adopt a more efficient snapshot-enabled storage system, such as Parallels [15].

### E. Network Reconfiguration

Another issue is that the VMs starting from the same VM snapshot will have the same network configuration (IP address,

MAC address, etc.) that is predetermined by the VM from which we take the snapshot. In the cloud, different VMs should have different network configurations. In our design, we take the following steps to achieve a VM's network reconfiguration. First, before the VM starts, we connect its virtual network interface card (NIC) to a private virtual bridge which prohibits all its outbound network communication. Then, through a virtual serial console, we pass a new network configuration into the VM and apply the changes in the VM. After the new network configuration takes effect, we re-connect the virtual NIC back to the normal virtual bridge which enables the VM to talk to the outside without any side-effect.

Unfortunately, network reconfiguration will cause problems for the applications running in the VM. For Internet services, there are two problems. First, it is sometimes impossible for an already initialized Internet service to adapt to a new IP address. To solve this problem, we make the service listen to an internal address, and build a DMZ (Demilitarized Zone) between this internal address and the VM's outbound address using iptables [16]. When the VM's outbound IP address changes, we only have to change the iptables' rule to adapt to the new address without reconfiguring the service itself. The other problem is that the database connection pool may reserve obsolete connections built before we take the snapshot. The server will be unresponsive for quite a long time while keeping on trying these connections. To cope with this problem, we configure JBOSS-like middlewares with the C3P0 connection pool, which detects connection failures via tentative SQL query and can reconnect to the database server after cumulative failures [17]. Besides, for the applications which implement a database connection pool itself, we can still hack its source code without much effort to implement the C3P0's mechanism to easily detect and recover from connection failures.

## V. IMPLEMENTATION

We have implemented a Twinkle prototype based on hardware-assisted virtual machine (HVM) technology of Xen 3.3.1 by modifying or adding 3,500 lines of source code. In this section, we will describe our implementation details, including VM snapshot preparation and fast startup.

### A. VM Snapshot Preparation

To prepare a VM snapshot, we first create a VM, boot the guest OS, and initialize and warm up the designated Internet service. Then, we take a VM snapshot based on this fresh VM, employing the *suspend* functionality of Xen. To support the techniques described in the previous section, we have modified Xen's suspend implementation as follows.

First, the structure of the execution snapshot is changed. To facilitate demand paging in Twinkle, the memory pages are relocated at the end of the execution snapshot, starting from an offset aligned on a page size boundary (e.g., 4K-byte). To locate demanded pages efficiently, we add *physical-to-disk* (p2d) table which maps the *physical page frame number* (pfn) of a memory page to its position on the storage. In addition, to assist free page avoidance, the hypervisor eliminates zero

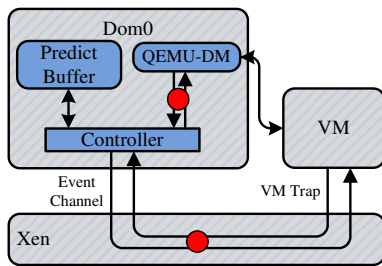


Fig. 5. The process of handling demand-page-traps.

pages in the VM snapshot, and mark it in the p2d table by setting the corresponding entry as -1. Third, after taking the VM snapshot, Xen continues to run the VM for an interval to track its working set, which is compressed by *gzip*.

### B. Fast Startup

For each VM, Dom0 (the management domain of Xen) creates a VM container, which takes as long as 3s-4s in our testbed. We observe that the time overhead comes mainly from virtual device creation. We are investigating optimizations to improve the series of operations. We can also reuse the VM containers in each physical server among different VMs to eliminate the time overhead, which will be our future work. In our current implementation, in order to reduce this impact, we create VM containers in advance.

When starting a VM, Dom0 first creates a COW image for the VM, fetches and uncompress the working set, and marks the remaining pages of the VM as *non-present* in the NPT. Then, Dom0 starts the VM execution. During the execution, when the guest OS attempts to access a non-present page, a demand-page-trap takes place. In an HVM domain, demand-page-traps are intercepted in three positions. First, when the guest OS accesses a non-present page with the memory access instructions, such as *MOV* and *CMP*, a demand-page-trap will be caught in the NPT page fault handler. Second, in a typical virtualization environment, the hypervisor traps sensitive instructions and emulates their executions. Some of these instructions, such as *LGDT* that loads a pseudo-descriptor from memory into the GDTR register, will also result in demand-page-traps if the referred pages are not present. These traps are handled in the sensitive instruction emulation handler. Third, the Xen hypervisor is not aware of any peripherals and reuses the Dom0’s drivers to manage devices. The hypervisor intercepts the guest OS’ I/O operations and delegates them to Dom0. For an HVM guest, Dom0 leverages *qemu-dm* to simulate I/O operations. When *qemu-dm* emulates a DMA operation and accesses a non-present page, a demand-page-trap will be intercepted in the I/O emulation routine.

Fig 5 shows the process of handling demand-page-traps. Since Dom0 is the management domain of Xen and manages hardware devices, most of our functional entities are located in Dom0. The principal entity is the *controller* kernel module which listens to demand-page-traps from Xen hypervisor and *qemu-dm* and fetches pages from the VM snapshot. The

*predict buffer* stores the predicted pages that have not been allocated to the VM.

When catching a demand-page-trap in Xen, Xen first allocates a zero page to the VM and updates the corresponding NPT entry. If the p2d entry is -1 (zero page), Xen hypervisor resumes VM execution at the trapped instruction immediately. Such a demand-page-trap is called *free page fault*. Otherwise, Xen hypervisor sends a notification to the *controller* through a designated event channel, requesting to fetch the referred page. On receiving this event, the *controller* first checks the *predict buffer*. If the page has been predicted, a *prediction hit fault* happens. The *controller* copies the page content into the newly allocated page. Otherwise, the page will be fetched from the VM snapshot, which is a *remote fetch fault*.

The handling process of the demand-page-traps from *qemu-dm* is similar to that of traps caught by the hypervisor, except that *qemu-dm* notifies the *controller* via a *proc* file system since *qemu-dm* runs in user space of Dom0 while *controller* is a kernel module. For brevity, we omit the introduction of the handling process of demand-page-traps caught by *qemu-dm*.

## VI. EVALUATION

In this section, we evaluate the performance of our prototype. Our experimental setup consists of seven Dell PowerEdge blade servers with Intel 5620 CPU, 24 GB RAM and 10K RPM SAS disks: one for Nginx [18] dispatchers, two elastic computing servers, one image storage server and three for a MySQL cluster. The architecture is shown in Fig 3, and all the servers are connected via a Gigabit switch. The two elastic computing servers are deployed with Twinkle to provide the mechanism of fast resource provisioning. The shared image storage is an iSCSI storage server holding the VM snapshots for the elastic computing servers. The Nginx dispatchers and the MySQL cluster are all running in a sufficient number of VMs on Xen 3.3.1, so that these components won’t become bottlenecks for either experiment. All the VMs, including the VMs that are started by Twinkle, are configured with 4096MB of RAM and one CPU. Both physical servers and VMs run Fedora 8 with kernel 2.6.18. To evaluate our system, we perform the following four experiments:

- 1 We show that Twinkle can start a VM fast without noticeable impact on the VM’s performance.
- 2 We use a single Internet service to clearly show that our system can respond to a flash crowd more rapidly compared to the existing approaches.
- 3 By starting two Internet services simultaneously, we demonstrate that in a shared virtualized environment, our system can still start multiple VMs efficiently.
- 4 We evaluate the effectiveness of our system in dealing with system faults by simulating a failure over.

### A. Workloads

We evaluate our system with a variety of benchmarks, including a subset of Spec CPU2006, RUBiS [6] and TPC-

W. Below, we will briefly describe these benchmarks:

1) *Spec CPU2006*: To evaluate the efficiency of our system, we use Spec CPU2006. As a relatively deterministic benchmark suite, it is suitable for comparing the performance among different techniques. Due to space constraints, we select six programs (*astar*, *libquantum*, *mcf*, *omnetpp*, *perlbench* and *xalancbmk*) from Spec CPU2006 based on [19].

2) *RUBiS*: RUBiS is used to simulate a bidding Internet service similar to EBay. In our experiment, we run each RUBiS instance in JBoss middleware configured with C3P0 connection pool to ease network reconfiguration. The data tier is deployed in the MySQL cluster with sufficient resources.

3) *TPC-W*: TPC-W is used to simulate an e-commerce Internet service similar to Amazon book store. In our experiment, each TPC-W instance runs in Tomcat. Like RUBiS, the data tier is deployed in the MySQL cluster. The implementation of its simple database connection pool is hacked to import the self-recovery mechanism of C3P0.

### B. Performance Evaluation

There are two important metrics related to our system, startup latency and performance overhead. Startup latency is defined as the time between the point when the VM starts to recover from a VM snapshot and the point when the guest OS begins to run. The startup latency of Twinkle is the time to fetch and decompress the working set. Performance overhead is evaluated by *VM invalid execution ratio* that is the percentage of CPU time when the VM is inactive. The higher the ratio is, the more performance degradation the VM will suffer.

To show our system’s effectiveness, we perform our experiment with the subset of Spec CPU2006. We run the six programs in separate VMs for two minutes, and then take a snapshot of each VM. After that, we continue the execution of the VM from the snapshot with three approaches: Xen’s native resume, demand paging without optimization, and our Twinkle startup. In the following, we will present the startup latency and the performance overhead of our system, and finally we will show how each technique contribute to the results.

To evaluate the startup latency, we compare Twinkle with Xen’s native resume (Baseline). For Xen’s native resume, it takes about 82.4 seconds to resume a VM with 4096MB of RAM. In contrast, the startup latency of Twinkle is proportional to the size of the VM’s working set. Table 1 presents the startup latency of each VM running different programs. The results indicate that Twinkle can start the VMs in less than 3.5 seconds except *mcf* which owns a large working set.

To evaluate its performance overhead, we record VM invalid execution ratio under three configurations throughout the execution of the six programs. Fig 6 shows our results (Y axis is in log scale). We present demand paging (worst case) and normal VM execution (best case, also the baseline) for comparison. As is shown in Fig 6, demand paging imposes heavy performance impact, especially at the beginning. In contrast, the invalid execution ratio under Twinkle is close to the best case in all the programs. It should be noted that the invalid execution

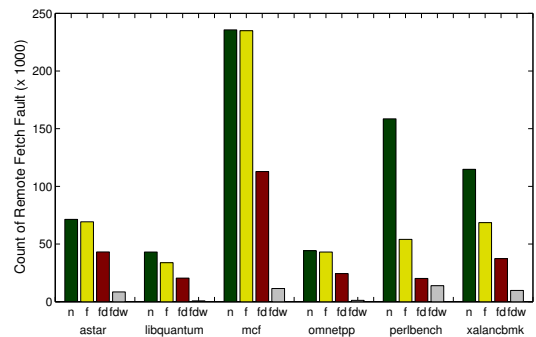


Fig. 7. The count of remote page faults under different combinations of techniques.

ratio of a normal VM is not zero due to the overhead coming from virtualization itself, which is trivial in our testbed. Table 2 presents the time to complete the execution of each program, as well as the slowdown relative to the baseline. We can see that the performance overhead of Twinkle is trivial for all the programs.

There are three types of page faults in our system: free page fault, prediction hit fault and remote fetch fault. The handling time of each is 0.6us, 14.3us and 398.6us, respectively. Obviously, the number of remote fetch fault tightly determines the performance of the newly started VM. We use the number of remote page fault to evaluate the contribution of each technique. We start the VM snapshot with the following combinations of techniques: *none* (“n”), *free page avoidance* (“f”), *free page avoidance + demand prediction* (“fd”) and *free page avoidance + demand prediction + working set estimation* (“fdw”). From Fig 7, we can see that the three programs, *libquantum*, *perlbench* and *xalancbmk*, benefit much from free page avoidance. With demand prediction enabled, the count of remote page fault is reduced by almost half in all programs because those pages are loaded in advance before they are actually accessed. With working set estimation integrated, the remote page faults decreases further.

### C. Application Evaluation

In this section, we evaluate Twinkle in providing additional VMs for Internet services that employ the auto scaling of cloud computing. Our testbed is constructed as Fig 3. In this paper, we focus on the fast resource provisioning mechanism. To evaluate this mechanism, in the dispatcher, we implement a heuristic policy of Scalr [20], which determines when to scale up services based on predefined policies. In our experiments, we adopt the policy of provisioning more VMs for the service when detecting degradation of service quality based on response time (exceeding 2000 msec in our cases). The dispatcher dynamically dispatches requests among all VMs employed by the service in a weighted round robin manner according to the response time from each VM.

In these experiments, we follow the way of Bodík et al [21] to generate workload spikes. We employ three approaches to increasing additional service instances: starting a VM from

TABLE I  
STARTUP LATENCY (SECONDS) AND RATIO TO BASELINE (PERCENTAGE).

	astar	libquantum	mcf	omnetpp	perlbench	xalancbmk
<b>Startup Latency</b>	1.49 (1.8%)	0.35 (0.4%)	11.89 (14.4%)	1.31 (1.6%)	2.25 (2.7%)	3.24 (3.9%)

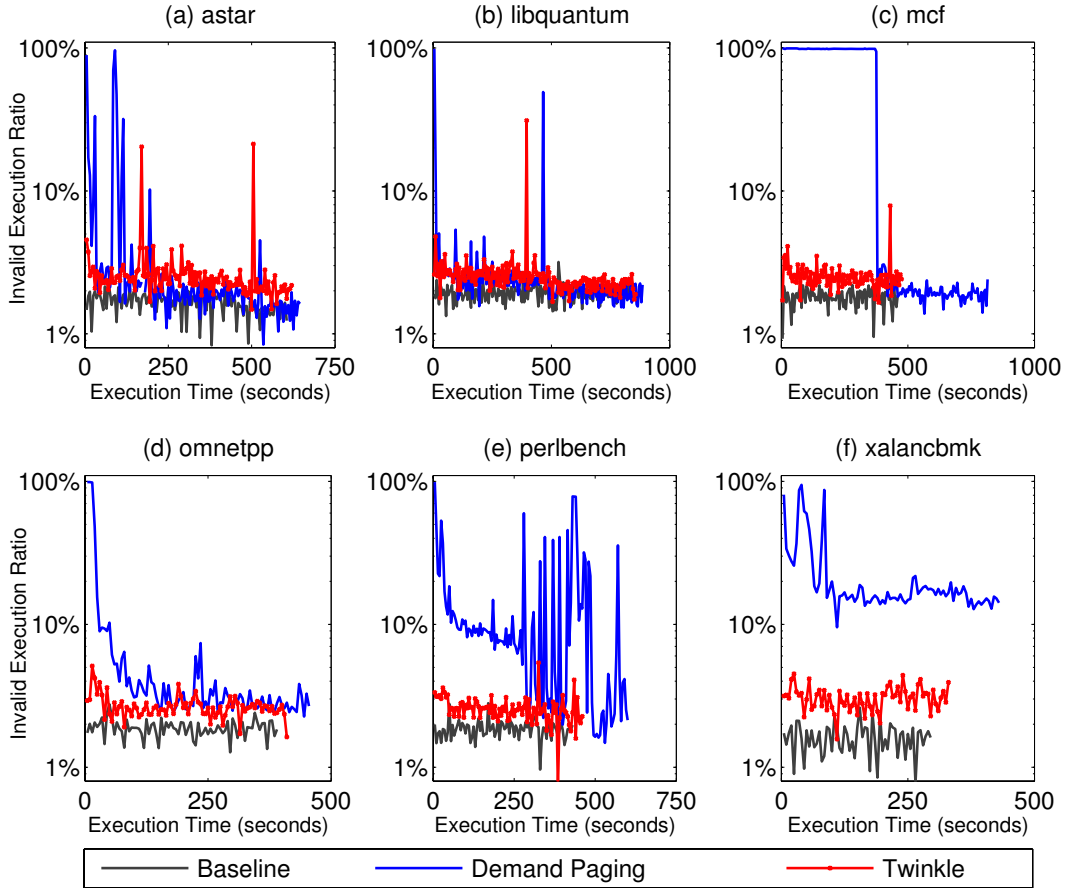


Fig. 6. VM Invalid Execution Ratio.

TABLE II  
EXECUTION TIME.

	astar	libquantum	mcf	omnetpp	perlbench	xalancbmk
<b>Baseline</b>	600.95	830.57	455.84	383.99	411.30	283.27
<b>Demand Paging</b>	639.56(6.4%)	863.28(3.9%)	818.93(79.7%)	459.74(19.7%)	575.04(39.8%)	425.34(50.2%)
<b>Twinkle</b>	608.31(1.2%)	841.57(1.3%)	470.15(3.1%)	399.75.04(4.1%)	460.40(11.2%)	325.71(11.5%)

scratch (“Xen VM Start”), resuming a VM snapshot with Xen’s native resume (“Xen VM Resume”) and starting from a VM snapshot with Twinkle (“Twinkle Fast Start”). With Twinkle, both RUBiS and TPC-W services begin processing client requests in less than 3 seconds after Scalr decides to provide additional service instances.

1) *Single Internet Service*: In this experiment, we use a single application, RUBiS, to show that Twinkle can maintain the application-level performance by quickly providing additional VMs when a flash crowd occurs. Initially, the number of RUBiS clients is 50, and one VM with 4096MB RAM and one CPU is sufficient to handle the requests. As shown in Fig

8, the response time (averaged in one second interval) in this period is low. After 40 seconds, we increase the number of clients from 50 to 450 in 400 seconds to simulate a workload spike. At roughly 300 seconds, our Scalr detects this flash crowd and starts an additional VM.

Fig 8(a) shows the results by starting a VM instance from scratch. The response time increases to over 3000 msec in some period, and it takes about 250 seconds before the scale-up decision takes effect. Fig 8(b) shows the performance by resuming a VM from a VM snapshot. Since the time to resume a VM is proportional to the size of RAM, it takes nearly 300 seconds before the response time decreases below 2000 msec.

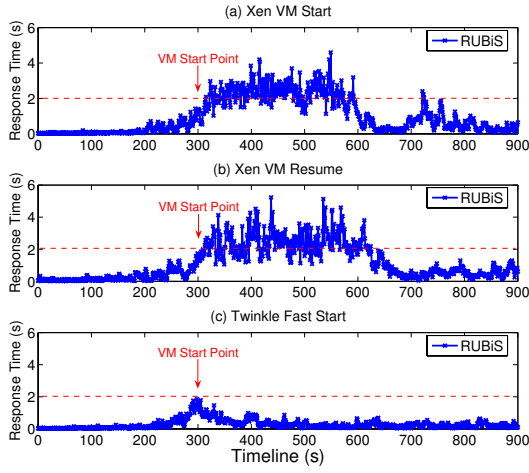


Fig. 8. Response Time of RUBiS service during a flash crowd.

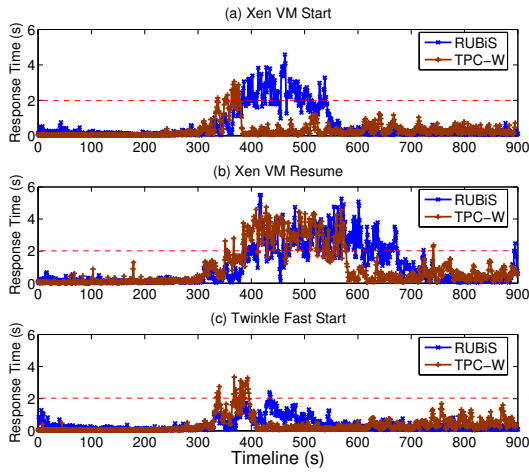


Fig. 9. Response Time of RUBiS and TPC-W services during a flash crowd.

It should be noted that this time span is longer than the latency of VM startup/resume time since it takes more time for a service to recover to normal if the service has been heavily overloaded. In Fig(c), we show the service quality achieved by Twinkle. With Twinkle, service capacity can be increased as soon as the Scalr detects a service degradation. As shown in Fig 8(c), the QoS in this test returns to normal quickly, keeping the response time below 2000 msec most of the time during the period of the flash crowd.

2) **Multiple Internet Services:** In the cloud, there are multiple services running in a single server. When multiple services are started simultaneously at a single server, they will race each other for resource. To illustrate our system in such an environment, we show the robustness of Twinkle when both RUBiS and TPC-W experience workload spikes simultaneously.

In the first part of this experiment, both RUBiS and TPC-W have 50 clients to generate workload. At the time of 50 seconds, we increase the number of TPC-W clients to 350 in 300 seconds, and at the time of 100 seconds, we increase the number of RUBiS clients to 300 in 200 seconds. Both services

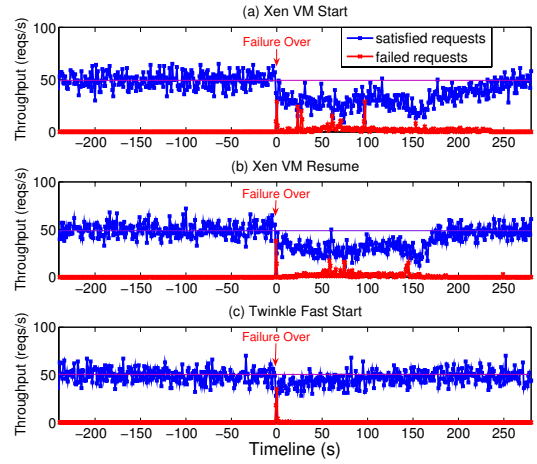


Fig. 10. Throughput of RUBiS services during a failure over.

will experience QoS degradation during the period, and will demand more capacity. Driven by the client requests, Scalr decides to increase VM instances for each service at almost the same time (within a difference of five seconds). In order to simulate the scenario where multiple services are started in a single server simultaneously, we start the two VMs at the same elastic computing server.

Fig 9 shows the results. Since TPC-W is a simple benchmark and runs in Tomcat that is lightweight, its startup latency is fast when starting from scratch (Fig 9(a)). When starting the two VMs simultaneously, there is little impact on both services, since our server has sufficient CPU and memory. However, in Fig 9(b), the TPC-W experiences more QoS degradation due to the long latency of fetching memory pages via network caused by Xen's native resume. In our testbed, we find if resuming the two VMs simultaneously, the resuming time of each VM increases to about 174 seconds due to the I/O bottleneck, while resuming a single VM takes about 83 seconds. In contrast, Twinkle consumes I/O resource not so greedily. From Fig 9(c), we can see that by fetching pages on demand, Twinkle still works well by starting new VMs in less than 3 seconds when provisioning resources among multiple services at the same server.

3) **High Available Internet Service:** In the cloud, when a VM crashes, we can quickly reprovision another VM from the infinite resource pool. In this experiment, we show that Twinkle handles service failures effectively as well.

Fig 10 demonstrates the degree of high availability achieved with different approaches to reprovisioning a VM. In this experiment, the number of RUBiS clients is fixed as 350. The average throughput is 50 requests per second since each client generates requests with a think time of 7 seconds. To sustain high availability and application level performance, there are two VMs serving these requests. At the time point of zero, we crashes one of the VMs to simulate a failure. Our dispatcher detects this failure and redirects all the requests to the other VM. Meanwhile, Scalr adds another VM replica as a recovery.

In this experiment, the latency of resource reprovisioning

determines the degree of high availability. Fig 10(c) shows that with our fast resource provisioning mechanism, the RUBiS server survives the failure much better. In contrast, when starting the VM from scratch (fig 10(a)) or resuming the VM from a VM snapshot (fig 10(b)), the RUBiS service experiences a severe throughput degradation.

## VII. RELATED WORK

Auto scaling, one of the most important feature of cloud computing, has attracted much interest from both academe and industry [2][3][22]. The commercial products, such as RightScale [23], Scalr [20] and EC2 itself [3], provide the ability of provisioning virtual machines on-the-fly to handle request spikes and decommissioning them when no longer needed. These products mainly focus on the policy decisions of when and where to start/stop virtual machines. For example, RightScale employs a simple democratic voting process to decide when scaling up/down should be taken. In this paper, we provide a fast resource provisioning mechanism which executes these decisions more efficiently once they are made.

Many projects have taken advantage of virtual machine replication to enable interesting applications. The Collective project [13] was designed to move the state of running virtual machines via a network. Collective aims at a low-bandwidth network and it begins execution after all the memory has been loaded, while Twinkle works in data centers with high bandwidth and fetches only the working set before starting the virtual machine. The Potemkin project [24], a honeypot system, simulating multiple virtual machines in a single server. It spawns short-lived virtual machines with memory copy-on-write techniques, and does not fetch memory pages via a network environment. Recently, the project of SnowFlock [25] enables cloning virtual machines on-the-fly for computation intensive applications. It employs multicast and an intrusive avoidance heuristics to reduce the pages transferred. Twinkle aims to improve the latency of virtual machine provisioning for the Internet services which employ the feature of auto scaling in the cloud. It works transparently without intruding the upper applications which are provided by cloud customers. Besides reducing startup latency, Twinkle also achieves low performance overhead at the beginning of startup, where the previous projects did not take much consideration.

## VIII. CONCLUSION

We present Twinkle, a fast resource provisioning mechanism for the Internet services which facilitate the feature of auto scaling in the cloud. By starting a virtual machine from partial snapshot and other techniques, we reduce the time to provision a virtual machine to a few seconds without noticeable performance overhead. With Twinkle, the Internet services can keep closer to capacity requirement and can maintain application level performance in the cases of flash crowds and failures.

## IX. ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work is supported by

the National Grand Fundamental Research 973 Program of China under Grant No.2007CB310900 and the MoE-Intel Joint Research Fund MOE-INTEL-09-06.

## REFERENCES

- [1] J. Elson and J. Howell, "Handling Flash Crowds from Your Garage," in *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX '08)*, 2008.
- [2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, Randy H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," *Technical Report No. UCB/EECS-2009-28, University of California, Berkeley*, 2009.
- [3] "Amazon EC2." [Online]. Available: <http://aws.amazon.com/ec2/>
- [4] W. LeFebvre, "CNN.COM: Facing a World Crisis," *Talk at 15th USENIX Systems Administration Conference*, 2002.
- [5] 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 ACM SIGOPS 19th Symposium on Operating Systems Principles (SOSP '03)*, vol. 37, no. 5, 2003.
- [6] "RUBiS." [Online]. Available: <http://rubis.ow2.org/>
- [7] AMD, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 2006.
- [8] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2009.
- [9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.
- [10] "S3." [Online]. Available: <http://aws.amazon.com/s3/>
- [11] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, p. 181, 2002.
- [12] Y. Li and Z. Lan, "A Fast Restart Mechanism for Checkpoint/Recovery Protocols in Networked Environments," in *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks (DSN '08)*, 2008.
- [13] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the Migration of Virtual Computers," *Proceedings of Operating Systems Design and Implementation (OSDI '02)*, 2002.
- [14] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX '05)*, 2005.
- [15] D. H. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. H. J. Feeley, N. C. Hut Hinson, and A. Warfield, "Parallax: Virtual Disks for Virtual Machines," in *Proceedings of the 2008 European Conference on Computer Systems (EuroSys '08)*, 2008.
- [16] "Netfilter." [Online]. Available: <http://www.netfilter.org/>
- [17] "C3P0." [Online]. Available: <http://sourceforge.net/projects/c3p0/>
- [18] "Nginx." [Online]. Available: <http://nginx.org/en/>
- [19] A. Phansalkar, A. Joshi, and L. John, "Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite," *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*.
- [20] "Scalr." [Online]. Available: <https://www.scalr.net/>
- [21] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, Modeling, and Generating Workload Spikes for Stateful Services," in *Proceedings of the first ACM Symposium on Cloud Computing (SOCC '10)*, 2010.
- [22] "Azure." [Online]. Available: <http://www.microsoft.com/windowsazure/>
- [23] "RightScale." [Online]. Available: <http://www.rightscale.com/>
- [24] 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," *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005.
- [25] H. A. Lagar-cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. D. Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing," in *Proceedings of the 2009 EuroSys Conference on Computer Systems (EuroSys '09)*, 2009.