Here Today, Gone Tomorrow: Exploiting Transient Servers in Data Centers

Rahul Singh[†], Prateek Sharma, David Irwin, Prashant Shenoy, and K.K. Ramakrishnan[‡]
University of Massachusetts Amherst [†]Nutanix [‡]Rutgers University

Abstract—Distributed applications implicitly assume that data center servers are available unless they fail. However, many emerging scenarios are altering this assumption by exposing *transient availability*, such that servers are available temporarily for an uncertain amount of time. *Transient servers* are cheaper and more energy-efficient than stable servers with continuous availability. In this paper, we highlight the characteristics of transient servers and then show how to exploit one particular characteristic: an advance warning of unavailability. Our system, called Yank, defines a bounded-time VM migration mechanism that leverages an advance warning of a few seconds to provide high availability cheaply and efficiently at large scales by enabling a backup server to maintain "live" snapshots for many transient VMs. Our experiments show that one backup server can concurrently support up to 15 transient VMs with minimal performance degradation with advance warnings as small as five seconds, even when VMs run memory-intensive interactive applications.

Index Terms—Transience, Virtualization, Migration, High Availability

1 Introduction

Due to the rising popularity of cloud computing, the number of data centers, and their size, continues to grow at a rapid pace.[1] Importantly, the distributed applications that run in data centers are generally built with the implicit assumption that the common case is for data center servers to be available—they may fail, but failures are uncommon, and when they happen, well-known techniques for replicating data and failing over to backup systems mitigate their performance impact [2]. However, many emerging scenarios are now altering this long-standing basic assumption. Rather than attempt to ensure continuous server availability and then design systems to mask rare failures, these scenarios instead offer *transient availability*, such that servers are available only temporarily for an uncertain amount of time. *Transient servers* are often cheaper and more energy-efficient than *stable servers*, which provide continuous availability. We briefly describe a few emerging scenarios that exhibit transient server availability.

Renewable Energy. Data centers are beginning to integrate local renewable energy sources into their power infrastructure. For instance, Apple's new iCloud data center in North Carolina uses power from a co-located 40MW solar farm, while Facebook is building a data center in Iowa that will run entirely on wind power from a nearby wind farm. Since the sun and wind generate power intermittently, data centers that rely on these energy sources may experience power fluctuations (shortages and surpluses) that require them to deactivate and re-activate servers as available power varies, resulting in transient server availability.

1. Portions of this paper appeared in a previous paper published at the USENIX NSDI conference in 2013 [1].

Smart Grids. Data centers may also interact with a "smart" electric grid to reduce their electricity bill. For example, in exchange for lower rates, demand-response programs enable utilities to signal data centers to curtail their power usage during periods of "peak" power demand in the grid, e.g., on hot summer days. As above, periodically curtailing power usage requires data centers to deactivate servers, and then reactivate them once the "peak" period ends, resulting in transient server availability.

Cloud Markets. Finally, data centers drive down the cost of computing by leveraging statistical multiplexing and massive economies of scale. To maintain near 100% utilization, cloud markets often sell any excess capacity to the highest bidder. Cloud markets are attractive to users due to their low prices, which are often 2x-10x less than the price to reserve a server and ensure it cannot be taken away. As an example, Amazon's Elastic Compute Cloud (EC2) operates a "spot" market for servers, where a server's hourly price fluctuates continuously based on market demand. Thus, if a spot server's market-determined price ever rises above the price a customer is willing to pay, then Amazon immediately takes it away. Such price fluctuations result in transient server availability due to unpredictable server allocations and revocations based on the spot price.

1.1 New Challenges

Handling transient server availability raises new challenges in systems design. While systems could simply mask transience by treating a newly unavailable server as a failure, and then employing traditional fault-tolerance techniques, we argue that these techniques are too costly for transient servers. Since transience arises from a desire to cut costs by relaxing the requirement for near-continuous server availability, employing expensive fault-tolerance techniques would eliminate transience's benefits. For example, a simple way to mask a server fault is to failover to a dedicated backup server that replicates the primary server's state [2]. However, maintaing a dedicated backup server for each transient server would increase power usage (in the first two scenarios) and rental costs (in the last scenario), or exactly the metrics each scenario was attempting to reduce. Thus, optimization techniques for transient servers must be lightweight to preserve transient servers' low cost.

Ultimately, handling transient server availability differs from handling failures in at least three ways, which leads to different systems design decisions.

High Churn Rate. Transient servers may "fail," i.e., become unavailable, at a much higher rate than conventional servers fail. In addition, unlike an actual failure, a transient server "failure" is often temporary: the server may automatically become available again at some point in the future, e.g., if available power or the spot price changes. Thus, transient-aware systems should gracefully handle high churn rates, with transient servers continuously leaving and joining the active set of servers.

Server Selection. While uncontrollable external conditions dictate *when* and *how many* transient servers are active at any time, transient-aware systems are often able to select *which* servers are active over time. For example, assuming there is enough available power to activate 25% out of a total of N servers, a system may choose to select $N/4$ servers and keep them continuously active, or, alternatively, activate a new set of $N/4$ servers (and deactivate the previous set) every T minutes. Cloud markets enable similar functionality by bidding different prices for different servers. Such freedom—to select which servers are available and unavailable over time—does not exist in conventional failure scenarios.

Advance Warning. Finally, techniques often exist to enable an advance warning before a transient server "fails" and becomes unavailable. For example, in EC2, whenever a server's spot price exceeds a consumer's bid price, the consumer has two minutes before the server terminates. As another example, Universal Power Supplies (UPSs) for racks provide some time after a power shortage before servers completely lose power. Thus, while conventional server failures occur without warning, transient servers may employ optimizations to exploit a brief advance warning.

We believe there is an opportunity to leverage transient servers' unique mix of characteristics—high churn rate, server selection, and an advance warning—to design low-cost techniques that mitigate transience's impact on performance. For instance, in prior work, we propose a blinking abstraction to handle high churn rates by rapidly selecting, i.e., activating and deactivating, new servers over time [3]. In this paper, we introduce a new technique—*bounded-time virtual machine (VM) migration*—to exploit an advance warning period.

2 Design Space

We assume a virtualized data center where applications run inside VMs on one of two types of physical servers: (i) always-on *stable servers* with a highly reliable power source and (ii) *transient servers* that may terminate anytime after an *advance warning* of size T_{warn}.

Once a transient server receives an advance warning, a data center must move any VMs (and their associated state) hosted on the transient server to a stable server to maintain their availability. Depending on T_{warn}'s duration, two possible solutions already exist to transition a VM to a stable server. If T_{warn} is large, it may be possible to live migrate a VM from a transient to a stable server. VM migration requires copying the memory and disk (if necessary) state [4], so the approach is only feasible if T_{warn} is long enough to accommodate the transfer. Completion times for migration are dependent on a VM's memory and disk size, with prior work reporting times up to one minute for VMs with only 1GB memory and no disk state [5].

The second approach is to employ a *high availability mechanism*, such as Remus [6], [7], which requires maintaining a live backup copy of each transient VM on a stable server. In this case, a VM transparently fails over to the stable server whenever its transient server terminates. While the approach supports warning times of zero ($T_{warn} = 0$), it requires the high runtime overhead of continuously propagating state changes from the VM to its backup. In some cases, memory-intensive, interactive workloads may experience 5X degradation in latency [6]. Supporting an advance warning of zero also imposes a high cost, requiring a backup server to keep VM memory snapshots resident in its own memory. In essence, supporting a warning time of zero requires a 1:1 ratio between transient and backup servers. Unfortunately, storing memory snapshots on disk is not an option, since it would further degrade performance by reducing the memory bandwidth ($\sim$3000MB/s) of primary VMs to the disk bandwidth ($<$100MB/s) of the backup server.

VM migration and high availability represent two extreme points in the design space for migrating transient servers. The former has low overhead but requires long warning times, while the latter has high overhead but handles warning times of zero. To exploit the middle ground between these two extremes, we propose a new mechanism for bounded-time VM migration, called Yank. Yank optimizes for modest advance warnings by maintaining a backup copy, similar to Remus, of each transient VM's memory and disk state on a stable backup server. However, Yank focuses on keeping costs low, by highly multiplexing each backup server across many transient VMs. As we discuss, our approach requires storing portions of each VM's memory backup on stable storage. We show that for advance warnings of a few seconds, Yank provides similar failover properties as high availability, but with an overhead and cost closer to

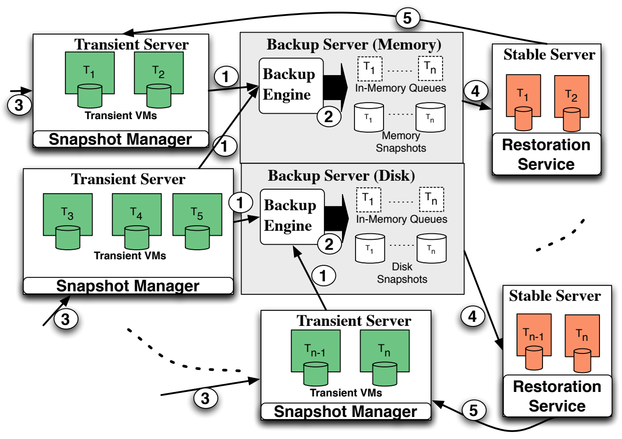

Fig. 1. Yank's Design and Basic Operation

VM migration. In fact, Yank devolves to high availability for a warning time of zero, and reduces to a simple VM migration as the warning time becomes larger.

3 Yank Design

Yank leverages the advance warning time to scale the number of transient servers independently of the number of backup servers by controlling when and how frequently transient VMs commit state updates to the backup server. In essence, the warning time T_{warn} limits the amount of data a VM can commit to its backup server after receiving a warning. Thus, during normal operation, a transient VM need only ensure the size of dirty memory pages and disk blocks remains below this limit. Maintaining this invariant guarantees that no update will be lost if a VM terminates after a warning, while providing additional flexibility over when to commit state. To keep its overhead and cost low, Yank highly multiplexes backup servers, allowing each to support many (>10) transient VMs by i) storing VM memory and disk snapshots, in part, on stable storage and ii) using multiple optimizations to prevent saturating disk bandwidth. Thus, given an advance warning, Yank supports the same failure properties as high availability, but uses fewer resources, e.g., hardware or power, for backup servers.

3.1 Yank Architecture

Figure 1 depicts Yank's architecture, which includes a *snapshot manager* on each transient server, a *backup engine* on each stable backup server, and a *restoration service* on each stable (non-backup) server. We focus primarily on how Yank maintains memory snapshots at the backup server, since we assume each backup server cannot keep memory snapshots for all transient VMs resident in its own memory. Thus, Yank must mask the order-of-magnitude difference between a transient VM's memory bandwidth (∼3000MB/s) and the backup server's disk bandwidth (<100MB/s). By contrast, maintaining disk snapshots poses less of a performance concern, since the speed of a transient VM's disk and its backup server's disk are similar in magnitude. This characteristic combined with a multi-second warning time permits asynchronous disk mirroring to a backup server without significant performance degradation.

Figure 1 also details Yank's functions. The snapshot manager executes within the hypervisor of each transient server and tracks the dirty memory pages of its resident VMs, periodically transmitting these pages to the backup engine, running at the backup server (1). The backup engine then queues each VM's dirty memory pages in its own memory before writing them to disk (2). Yank assumes an external service, e.g., a power manager or cloud platform, provides advance warning notifications for transient servers (3). The service both i) informs backup and transient servers when the warning time changes and ii) issues warnings to transient and backup servers of an impending termination. Since Yank depends on warning time estimates, the service above must run on a stable server.

Upon receiving a warning (3), the snapshot manager pauses its VMs and commits any dirty pages to the backup engine before the transient server terminates. The backup engine then has two options, assuming it is too resource-constrained to run VMs itself: either store the VMs' memory images on disk for later use (for batch applications), or migrate the VMs to another stable (non-backup) server (for interactive applications) (4). In the latter case, we assume an exogenous placement policy exists to select a destination stable server in advance to run a transient VM if its server terminates. One example placement policy for a data center subject to variations in available power might be to spread transient VMs across as many physical servers as possible when power is plentiful or cheap to maintain reserve capacity to instantaneously handle unexpected flash crowds. However, whenever power becomes scarce or expensive, the data center does not have the luxury of such reserve capacity and must quickly (within the warning time) consolidate transient VMs onto a smaller set of stable servers. Note that in some cases, e.g., as with severe power shortages, there may not be enough stable servers to adequately satisfy an application's workload. Defining and experimenting with specific placement policies for the scenarios in Section 1 is outside the scope of this paper.

Finally, Yank executes a simple restoration service on each stable (non-backup) server to facilitate VM migration and restoration after a transient server terminates. Yank currently copies a VM's entire memory image from the backup server to the destination server before restoring it. The restoration time is a function of the network bandwidth and the size of the memory image, and is equivalent to a typical stop-and-copy VM migration. Supporting live restoration using either a pre-copy [4] or post-copy [8] migration is the subject of future work.

3.2 Just-in-Time Synchrony

Since Yank receives an advance warning of time T_{warn} before a transient server terminates, its VM memory snapshots stored on the backup server need not maintain strict external synchrony [9]. Instead, upon receiving a warning of impending termination, Yank only has to ensure what we call *just-in-time synchrony*: a transient VM and its memory snapshot on the backup server are always capable of being brought to a consistent state before termination. To guarantee just-in-time synchrony, as with external synchrony, the snapshot manager tracks dirty memory pages and transmits them to the backup engine, which then acknowledges their receipt. However, unlike external synchrony, just-in-time synchrony only *requires* the snapshot manager to buffer a VM's externally visible, e.g., network or disk, output when the size of the dirty memory pages exceed an upper threshold U_t, such that it is impossible to commit any more dirty pages to the backup engine within time T_{warn}.

In the worst case, with a VM that dirties pages faster than the backup engine is able to commit them, the snapshot manager is continually at the threshold, and Yank reverts to high availability-like behavior by always delaying the VM's externally visible output until its memory snapshot is consistent. Since we assume the backup server is not able to keep every transient VM memory snapshot resident in its own memory, the speed of the backup engine's disk limits the rate it is able to commit page changes. While memory bandwidth is an order of magnitude (or more) greater than disk (or network) bandwidth, Yank benefits from well-known characteristics of typical in-memory application working sets to prevent saturating disk (or network) bandwidth. Specifically, the size of in-memory working sets tend to i) grow slowly over time and ii) be smaller than the total memory [10].

Slow growth stems from applications frequently rewriting the same memory pages, rather than always writing new ones. Yank only has to commit the last rewrite of a dirty page (and not the intervening writes) to the backup server after reaching its upper threshold of dirty pages U_t. In contrast, to support termination with no advance warning, a VM must commit nearly *every* memory write to the backup server. In addition, small working sets enable the backup engine to keep most VMs' working sets in memory, reducing the likelihood of saturating disk bandwidth from writing dirty memory pages to disk. Recent work extends this insight to collections of VMs in data centers, showing that while the size of a single VM's working set may experience temporary bursts in memory usage, the bursts are often brief and not synchronized across VMs [11]. Yank relies on these observations in practice to highly multiplex each backup server's memory without saturating disk bandwidth or degrading transient VM performance during normal operation.

4 Yank Implementation

We implement Yank's snapshot manager by extending Remus inside the Xen hypervisor (v4.2). By default, Remus tracks dirty pages over short epochs ($\sim$100ms) using shadow page tables and pausing VMs each epoch to copy dirty memory pages to a separate buffer for transmission to the backup server. While VMs may speculatively execute after copying dirty pages to the buffer, but before receiving an acknowledgement from the backup server, they must buffer external network or disk output to preserve external synchrony. Remus only releases externally-visible output from these buffers after the backup server has acknowledged receiving dirty pages from the last epoch. Rather than commit dirty pages to the backup server every epoch, our snapshot manager uses a simple bitmap to track dirty pages and determine whether to commit these pages to backup engine based on the upper and lower threshold, U_t and L_t. In addition, rather than commit CPU state each epoch, the snapshot manager only commits CPU state when it receives a warning that a transient server will terminate.

We implement Yank's backup engine using a combination of Python and C, with a Python front-end that accepts network connections and forks a backend C process for each transient VM. For each transient VM, the backend C process accepts dirty page updates from the snapshot manager and sends acknowledgements. Each update includes the number of pages in the update, as well as each pages' page number and contents (or a delta from the previous page sent). The process then places each update in an in-memory producer/consumer queue that removes pages in-turn—in Least Recently Used (LRU) order based on a timestamp—and writes them to disk. In the current implementation, the backend process stores VM memory pages sequentially in a file on disk. For simplicity, the file's format is the same as Xen's format for storing saved VM memory images, e.g., via `xm save`.

Finally, we implement Yank's restoration service at user-level in C. The daemon accepts a VM memory snapshot and an in-memory queue of pending updates, and then applies the updates to the snapshot without writing to disk. Since our implementation uses Xen's image format, the service uses `xm restore` from the resulting in-memory file to re-start the VM.

5 Experimental Evaluation

We evaluate Yank's VM performance, downtime after a warning, and scalability. While our evaluation does not capture the full range of dynamics present in a production data center, it does demonstrate Yank's flexibility to handle a variety of dynamic operating conditions.

5.1 Experimental Setup

We run our experiments on 20 blade servers with 2.13 GHz Xeon processors with 4GB of RAM connected to

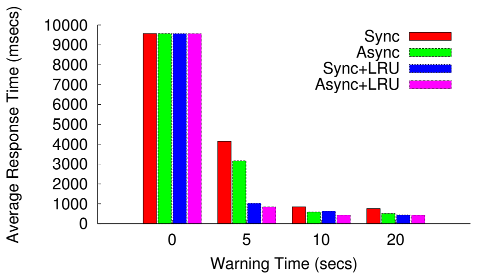

Fig. 2. TPC-W response time as warning time varies.

the same 1Gbps Ethernet switch. Each server running the snapshot manager uses our modified Xen (v4.2) hypervisor, while those running the backup engine and the restoration service use unmodified Xen (v4.2). In our experiments, each transient VM uses one CPU and 1GB RAM, and runs the same OS and kernel (Ubuntu 12.04, Linux kernel 3.2.0). Here, we focus on the TPC-W benchmark to stress Yank's ability to handle an interactive application that exhibits rapid writes to memory. TPC-W emulates an online store akin to Amazon. We use a Java servlets-based multi-tiered configuration of TPC-W that uses Apache Tomcat (v7.0.27) as a front end and MySQL (v5.0.96) as a database backend. TPC-W allows us to measure the influence of Yank on the response time perceived by the clients of an interactive web application.

5.2 Benchmarking Yank's Performance

Transient VM Performance. We first evaluate Yank's effect on VM performance during normal operation. In this experiment, transient VMs have a static warning time. We also limit the backup engine to using an in-memory queue of 300MB to store memory updates from each 1GB transient VM. We run each experiment for 15 minutes before issuing a warning to the transient and backup server, and evaluate its performance during the pre-warning period. Here, we focus on TPC-W, since it is an interactive application that is sensitive to VM pauses from buffering network or disk output. We measure the average response time of TPC-W clients, while varying both the warning time and the snapshot manager's policy for committing pages to the backup engine. Figure 2 shows that an asynchronous policy, which also selects pages to commit using LRU replacement, results in the lowest average response time. The asynchronous policy is an optimization that reduces VM pauses, because the snapshot manager begins committing pages to the backup as soon as it hits the lower threshold L_t rather than waiting until reaching U_t (as in a synchronous policy), and forcing a large commit to the backup server. The experiment also demonstrates that with even a brief five second warning, the response time is <500ms using the asynchronous policy with LRU.

By contrast, with a warning time of zero the average response time rises to over nine seconds. In addition, the 90^{th} percentile response time was also near 15 seconds, indicating that the average response is not the result of a few overly bad response times. With no warning, the VM must pause and mirror every memory write to the backup server and receive an acknowledgement before proceeding. Although Remus [6] does not use our specific TPC-W benchmark, their results with the SPECweb benchmark are qualitatively similar, showing 5X worse latency scores. Thus, our results confirm that even modest advance warning times lead to vast improvements in response time for interactive applications.

Result: *Yank imposes minimal overhead on TPC-W during normal operation. With a brief five second warning time, the average response time of TPC-W is 10x less (<500ms) than with no warning time (>9s).*

Scalability. The experiments above focus on performance with a single VM. We also evaluate how many transient VMs the backup engine is able to support concurrently, and the resulting impact on transient VM performance during normal operation. In this case, our experiments last for 30 minutes using a warning time of 10 seconds, and scale the number of transient VMs running TPC-W connected to the same backup server. We measure CPU and memory usage on the backup server, as well as the average response time of the TPC-W clients. Figure 3 shows the results, including the maximum of the average response time across all transient VMs observed by the TPC-W clients, the CPU utilization on the backup server, and the backup engine's memory usage as a percentage of total memory. The figure demonstrates that, in this case, the backup server is capable of supporting as many as 15 transient VMs without the average client response time exceeding 700ms. Note that without using Yank the average response time for TPC-W clients running our workload is 300ms. In addition, even when supporting 15 VMs, the backup engine does not completely use its entire CPU or memory.

Result: *Yank is able to highly multiplex each backup server. Our experiments indicate that with a warning time of 10 seconds, a backup server can support at least 15 transient VMs running TPC-W with little performance degradation for even a challenging interactive workload.*

6 Related Work

Yank's bounded-time VM migration mechanism is useful for any scenario that exhibits transient server availability. Most prior work encounters transient servers in the context of data center power management. For instance, supporting renewable energy sources in data centers often targets simple non-interactive batch jobs, since these jobs are tolerant to delays when renewable power is not available. Here, batch job schedulers [12], [13] can use predictions of future energy harvesting to align job execution with periods of high renewable generation. In contrast, since Yank's snapshots of memory and disk state are generic, it is capable of transparently supporting both batch and interactive applications.

Yank is also useful for cheaply scaling high availability in data centers where the majority of fail-stop failures

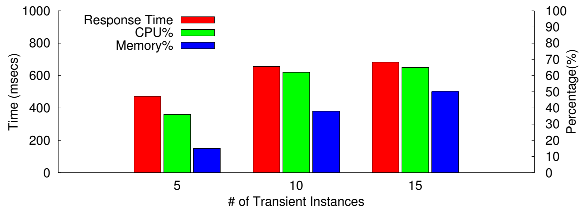

Fig. 3. Yank scalability

are power outages. Prior approaches to decreasing the cost of a data center's power delivery infrastructure have focused on maintaining its reliability, e.g., by using a mix of energy storage technologies [14]. Yank enables data centers to instead partially decrease their power infrastructure's reliability, e.g., using less UPS capacity, while still ensuring services remain available during outages. As a recent study shows, among all techniques for handling data center power outages, e.g., CPU throttling, sleeping, hibernation, VM migration, etc., Yank's approach incurs the lowest cost and has the highest performance for a representative application [15].

7 Conclusion

In this paper, we introduce the abstraction of a transient server, which may terminate anytime after an advance warning. We then design Yank with a bounded-time VM migration mechanism to exploit the advance warning. Yank fills the void between Remus, which requires no advance warning, and live VM migration, which requires a lengthy advance warning, to cheaply and efficiently support transient servers at large scale. In particular, our results show that a single backup server is capable of maintaining memory snapshots for up to 15 transient VMs with little performance degradation, which dramatically decreases the cost of providing high availability relative to existing solutions.

References

[1] R. Singh, D. Irwin, P. Shenoy, and K.Ramakrishnan, "Yank: Enabling Green Data Centers to Pull the Plug," in *NSDI*, April 2013.
[2] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, "Distributed Systems," ch. The Primary-Backup Approach, pp. 199–216, ACM Press/Addison-Wesley Publishing Co., 1993.
[3] N. Sharma, S. Barker, D. Irwin, and P. Shenoy, "Blink: Managing Server Clusters on Intermittent Power," in *ASPLOS*, March 2011.
[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *NSDI*, May 2005.
[5] H. Liu, C. Xu, H. Jin, J. Gong, and X. Liao, "Performance and Energy Modeling for Live Migration of Virtual Machines," in *HPDC*, June 2011.
[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *NSDI*, April 2008.
[7] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, "RemusDB: Transparent High Availability for Database Systems," in *VLDB*, August 2011.
[8] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy Live Migration of Virtual Machines," *SIGOPS Operating Systems Review*, vol. 43, July 2009.
[9] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the Sync," in *OSDI*, November 2006.
[10] P. Denning, "The Working Set Model for Program Behavior," *CACM*, vol. 26, Jan. 1983.
[11] D. Williams, H. Jamjoom, Y. Liu, and H. Weatherspoon, "Overdriver: Handling Memory Overload in an Oversubscribed Cloud," in *VEE*, 2011.
[12] I. Goiri, K. Le, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "GreenSlot: Scheduling Energy Consumption in Green Datacenters," in *SC*, April 2011.
[13] I. Goiri, K. Le, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "GreenHadoop: Leveraging Green Energy in Data-Processing Frameworks," in *EuroSys*, April 2012.
[14] D. Wang, C. Ren, A. Sivasubramaniam, B. Urgaonkar, and H. Fathy, "Energy Storage in Datacenters: What, Where, and How Much?," in *SIGMETRICS*, June 2012.
[15] D. Wang, S. Govindan, A. Sivasubramaniam, A. Kansal, J. Liu, and B. Khessib, "Underprovisioning Backup Power Infrastructure for Datacenters," in *ASPLOS*, March 2014.