Virtual Machine Reset-Atomicity in Xen

Adam C Everspaugh
University of Wisconsin-Madison
ace@cs.wisc.edu

Benita Bose
University of Wisconsin-Madison
bbose@wisc.edu

Abstract

We present a general solution to virtual machine (VM)-reset security vulnerabilities. By modifying the hypervisor and guest operating system, our solution guarantees *VM-reset atomicity* with the help of software transactional memory. We provide a general library function in C that effectively secures VM-reset-sensitive application code and demonstrate our prototype implementation on Xen-Linux using GCC's transactional memory support. Our prototype implementation adds only 54% execution-time overhead to typical encryption and digital signature operations.

1 Introduction

1.1 Background

Most virtualization platforms, including VMware, VirtualBox, KVM, and Xen support full state snapshots [5][12]. This is the ability to capture the entire state of the virtual machine: disk, registers, and memory; to a file. Full state snapshots have many uses including migration [8], crash recovery [7], fast boot, and VM cloning [13]. Unfortunately, full state snapshots are a relatively new concept in the world of computing and most application security code was written without considering the impact of resetting a machine from a snapshot. For the purposes of this paper, we define the term *VM-reset* (or just *reset*) to be the event when a VM is restored from a snapshot.

1.2 Motivation

In [18], Ristenpart, et al show that Firefox and Apache are vulnerable to VM-reset attacks. These applications reuse random numbers cached in memory for transport-layer security (TLS) connections allowing a remote attacker to defeat the security of TLS. Even worse, when the DSA signature algorithm is used, [18] demonstrates that an attacker can derive an Apache web server's *private key* when exploiting this VM-reset vulnerability. Further compounding this problem, no solutions have been proposed to address VM-reset vulnerabilities generally.

The most likely sources for vulnerabilities involve cryptography and security-related code. One-time-use values are common in cryptography and security protocols and are often necessary for security proofs to hold. One-time-use values include random numbers, initialization vectors (IVs) and counters for encryption, seeds for digital signature, and one-time passwords. Reuse of any of these sensitive values can result in an immediate, catastrophic loss of security [2]. Additionally, by design, applications are unaware that a snapshot has been taken or that a reset has occurred [22].

VM-resets only expose security vulnerabilities when the reset occurs more than once from the same snapshot. Because of this, VM migrations are unlikely to cause problems since a migration snapshot is typically used only once. However, fast boot, cloning, and crash recovery resets may expose VM-reset vulnerabilities in the guest OS or applications.

1.3 Threat Model

Our solution secures a target application under the threat model where the platform is trusted and there is a remote attacker. We assume that the hardware, hypervisor, and OS are trusted and reasonably secure. A remote attacker may observe any outputs of the target application. The attacker may interact with the application through it's normal API. For the threat model to be relevant, the application must have a VM-reset security vulnerability and resets must occur more than once from the same snapshot. We assume that snapshots are taken at *adversarially* chosen points in the target application's execution.

For a reasonable, real-world example, consider a ma-

chine hosting an Apache web server on an infrastructure-as-a-service (IaaS) provider like Amazon's Elastic Compute Cloud (EC2). Assume the VM is booted, Apache started, and a full-state snapshot is taken to rapidly provision new servers in response to demand. When new VMs are restored from this snapshot, a remote attacker can initiate TLS connections with the web server and observe the ServerHello.random values that the web server uses when negotiating connections. As shown in [18], reuse of the ServerHello.random with DSA enables an attacker to derive the server's private key.

2 Related Work

[9] first identified the possibility for VM-reset vulnerabilities with sensitive, one-time-use values, but didn't propose any solutions. [18] demonstrated the first VM-reset vulnerabilities in Firefox and Apache and showed that both VMware and VirtualBox are vulnerable platforms. [2] and [18] proposed using hedged cryptography as a solution. Hedged cryptography increases an attack's complexity but doesn't completely mitigate the problem and doesn't solve the general problem of VM-reset vulnerabilities. [3] points out that most cryptography is brittle in the face of bad (including reused) random numbers and can lead to catastrophic loss of security.

Our solution makes use of software transactional memory, which has been a very active area of research, including work by [20] and [11]. [15] discusses using transactional memory to guarantee atomicity.

3 Solution

We present a general solution to VM-reset vulnerabilities that provides hypervisor and OS support to userland applications so they can detect when a reset has occurred. Our solution also provides a userland library that uses software transactional memory to guarantee that sensitive code executes atomically with respect to a VM-reset. That is, we can guarantee that when a reset-sensitive transaction is committed, that no VM-reset occurred during the transaction. We call such a feature *reset-atomicity* and say it offers a *reset-atomicity guarantee*.

Our implementation uses the atomic transaction feature provided by GCC [14][17][19]. This new feature of GCC speculatively executes code marked as a transaction and detects read-write or write-write memory conflicts between two or more transactions. If conflicts are detected, the transaction is *canceled* and memory updates are rolled back. If no conflicts occur, the transaction is *committed* and memory updates persist. Originally designed to improve the ease of writing concurrent, multithreaded applications, we've found that the transaction cancel and commit feature simplifies adding reset-atomicity to an application. If reset-atomicity is violated, our solution simply cancels the transaction (undoing any memory effects) and retries the transaction. We repeat this process endlessly until the transaction succeeds under the assumption that reset events are quite rare.

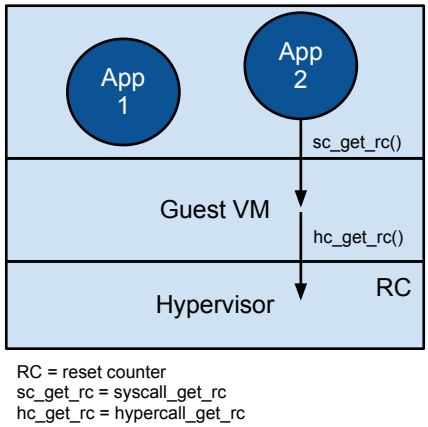

Figure 1: Solution architecture

3.1 Design

Figure 1 shows the design of our solution. We modified the Xen hypervisor [1] to keep a *reset counter* in memory that is incremented each time a guest VM is reset from a snapshot. We added a hypercall to Xen that allows the guest VM to access the current reset counter. We modified the Linux kernel to add a system call that will make this hypercall on behalf of userland applications to simplify the interface for userland applications. However, this hypercall is not a privileged operation and any application can make the hypercall directly if it desires.

Our solution includes a small library compiled with GCC's software transactional memory (TM) support [19]. We provide a single routine (figure 2) that guarantees reset-atomicity to any function executed with this routine. That is, if a reset occurs before the entire routine completes, any *revocable* side-effects of the routine are undone (the transaction is canceled) and the routine is re-executed until reset-atomicity is obtained.

GCC's TM distinguishes between *revocable* and *irrevocable* actions. Revocable actions are memory updates that occur in code compiled with GCC TM support enabled (`-fgnu-tm`). Irrevocable actions are defined as any system calls (or code that may eventually call a system call) or calls to library routines that were not themselves compiled with GCC TM support. While irrevocable ac-

```
void run_reset_atomic((void* func)(void))
{
  bool commit = false;
  while (!commit)
  {
    transaction_atomic
    {
      int my_rc = sys_get_rc();
      func();
      if (my_rc != sys_get_rc())
        transaction_cancel;
      else commit = true;
    }
  }
}
```

Figure 2: Pseudocode of reset-atomic wrapper function

tions (such as IO) cannot be undone, they can be *redone*. For instance, one of our prototype workloads reads random numbers from the Linux random number generator, `/dev/urandom`, using the `fopen` system call. If a transaction is canceled, this read cannot be undone. However, when the transaction is retried, a new random number is read and replaces the old value in memory. In many cases, re-doing an operation is a sufficient substitute for undoing the operation.

4 Analysis

We analyze our reset-atomic prototype for two features: correctness and performance. For correctness, we demonstrate that routines that are interrupted by a VM-reset will have memory side-effects undone and then are re-executed. To do this, we develop a simple routine and run it on a guest VM both with and without our reset-atomic wrapper. While the routine executes, we take snapshot from Domain 0 and then restore the guest VM from this snapshot. We show side-by-side output that confirms that the reset-atomic wrapper properly detects the VM-reset, rolls back memory, and re-executes the routine.

For performance analysis, we seek to determine how much overhead our solution contributes to expected workloads and to understand the source of this overhead. Our solution adds 2 system calls, 2 hypercalls, and transactional memory overhead to selected routines. We benchmark 3 workloads under a variety of conditions:

- No modifications (baseline execution time);
- Inside of an atomic transaction (but no system calls or hypercalls);
- Inside an atomic transaction with 2 nop system calls (but no hypercalls);
- Within reset-atomic wrapper (atomic transaction, 2 system calls, and 2 hypercalls).

To ensure that our *nop* system call adds no additional overhead (like an unintended VM exit), we added our own dummy system call to the Linux kernel so we could ensure that that system call was indeed a nop.

We choose three workloads for this performance analysis:

- empty - A nop function call
- nqueens - A large number of memory operations solving the nqueens problem for an 8×8 chessboard
- encryptSign
  - Read 2 random values from the Linux random number generator (`/dev/urandom`).
  - Encrypt a single block (128-bits) of data with AES-128 in CFB mode with a 128-bit random initialization vector.
  - Sign the resulting single block of ciphertext with DSA using a 1024-bit signing key and a 128-bit random seed.

EncryptSign performs all cryptographic operations with OpenSSL cryptographic library and is representative of code that requires reset-atomicity to ensure security.

We also wanted to understand any performance difference between committing and canceling an atomic transaction. To analyze this, we execute a single workload inside an atomic transaction and measure execution time with two different treatments: cancelling and committing the transaction.

5 Methodology

We developed, tested, and analyzed our reset-atomic prototype on a machine with a dual-core Intel Xeon E5430 CPU clocked at 2.7 GHz, 6 MB of L2 cache, hyper-threading disabled, and 14 GB of main memory. We used Xen 4.2.1 compiled from source with our own modifications for tracking resets and adding a hypercall. Domain 0 was configured with Fedora 18 and we ran a single guest VM with Ubuntu 12.10 configured with Linux kernel 3.5.7.11 compiled from source with our custom system calls added. Snapshot and restore operations (VM-resets) for our guest VM were performed on Domain 0 and all workloads were run on the Ubuntu guest VM. We used GCC 4.7.2 (from the package provided with Ubuntu

```
void test_function()
{
  static int counter = 0;
  printf("START counter: %d\n", counter);
  for (i = 1..100)
    ++counter;
  sleep(10);
  printf("END counter: %d\n\n", counter);
}
```

Figure 3: Pseudocode for correctness experiment

12.10) with the `-fgnu-tm` flag to enable transactional memory support. Only application code was compiled with GCC TM support - neither Xen nor the Linux kernel required changes to their default build processes.

For system calls and other irrevocable library functions in application code, we marked these calls with the GCC TM attribute `__transaction_pure`, to signal to GCC that these functions contain either no side effects or irrevocable side effects only. Revocable code was marked with the attribute `__transaction_safe` and the GCC compiler uses static analysis to confirm that these routines call only other functions marked as `safe` or `pure`.

For each performance experiment we run the workload once without timing to exclude any cache warm-up effects from the measurements. We then execute each workload 10 times under the specified treatment and report the median execution time.

We used `clock_gettime` for execution timing.

6 Evaluation

6.1 Correctness

Figure 3 shows the pseudocode for our correctness experiment and figures 4 and 5 demonstrate that our implementation is correct. As shown in figure 3, we added a 10 second sleep to our function-under-test in order to permit us to reliably perform a VM reset when the counter value reached 200. Note that in figure 5 because the print occurs inside the transaction (and because print is an irrevocable system call) the counter value 300 is briefly observed, but then it is rolled back to 200 when the transaction is canceled and re-tried.

Note The line `==== VM Reset ====` that appears in figures 4 and 5 is not program output. This line is inserted into each figure to designate when a VM-reset was performed from Domain 0.

```
START counter: 0
END   counter: 100

START counter: 100
END   counter: 200

START counter: 200
==== VM reset ====
END   counter: 300

START counter: 300
END   counter: 400
```

Figure 4: Output without atomicity guarantee

```
START counter: 0
END   counter: 100

START counter: 100
END   counter: 200

START counter: 200
==== VM reset ====
END   counter: 300

START counter: 200
END   counter: 300
```

Figure 5: Output with atomicity guarantee

6.2 Performance

Table 1 and figure 6 show the results of our performance measurements using median execution times. Clearly, GCC's transactional memory implementation adds significantly more overhead to the nqueens workload than the other workloads. This is because the current TM implementation makes a private copy of each variable in a transaction and tracks all reads and writes in the transaction in order to detect conflicts [4]. For our purposes, we don't anticipate conflicting transactions - instead we're only using the convenient feature that TM can *undo* a transaction when canceled. A streamlined implementation could simply create a private copy of the initial memory values when a transaction begins and then restore the initial values if the transaction is canceled. This would sidestep the high cost of tracking all reads and writes within a transaction.

High TM overhead in this version of GCC was also demonstrated in [4]. The author quotes the GCC developers: "The support is experimental ... several parts of the implementation are not yet optimized." Presumably, future versions of GCC will reduce TM overhead.

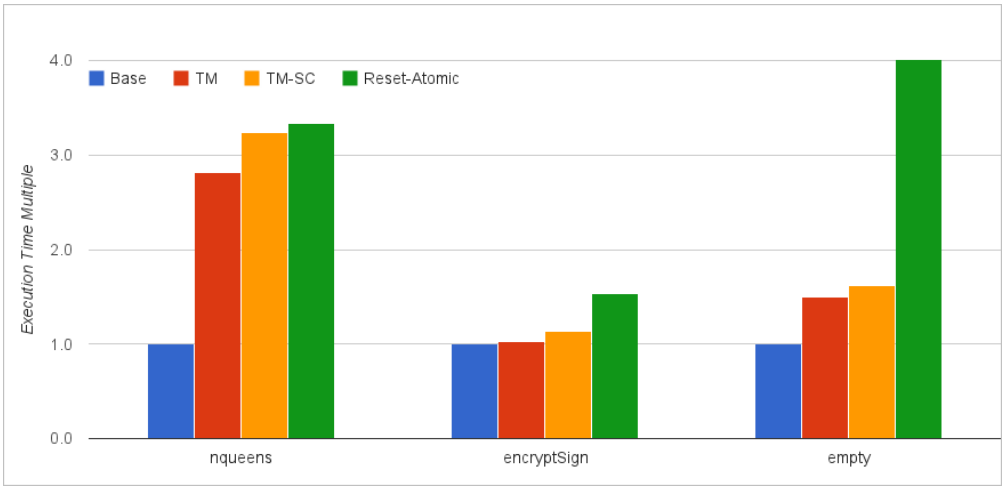

Figure 6: Workload performance relative to *base*

Workload	Base	TM	TM-SC	RA
nqueens (μs)	30.5	86.1	99.0	101.8
encryptSign (ms)	108.3	110.8	123.2	166.7
empty (μs)	1.3	1.9	2.1	17.3

Base	Workload with no modifications
TM	Workload + atomic transaction only
TM-SC	Workload + transaction + 2 NOP syscalls
RA	Workload + reset-atomic wrapper

Table 1: Median execution time for workloads

As seen in figure 6, the performance multiple for encryptSign is 1.54, with a significant amount of that overhead attributed to the hypercall. This overhead is significant, but not surprising: [10] points out that the time required to process VMExits (which occur during a hypercall) make up a significant portion of virtualization overhead. In the case of encryptSign, there are very few memory accesses, so TM overhead is minimal (approximately 2%).

Commit vs Cancel Figure 7 shows the difference between committing and canceling a transaction that encapsulates the encryptSign workload. The transaction adds a small overhead to encryptSign, but there is no discernible difference between committing or canceling a transaction under this workload.

6.3 Limitations

Our current solution tracks the reset counter in the main memory of the hypervisor, however, snapshot files persist on disk. If the hypervisor is restarted the reset counter returns to 0 and a VM-reset may go undetected. So our current implementation only provides reset-atomicity when the machine isn't restarted. Note that a monotonically incrementing reset counter is sufficient but not required. As long as the reset indicator changes in some way between resets, an application will detect the reset. A complete implementation could substitute an in-memory counter for any of the following:

- Timestamp - if wall clock time can be guaranteed;
- Hardware counter - if monotonicity can be guaranteed; or
- Random number generator - if available to the hypervisor.

Alternatively, instead of polling the OS and hypervisor for a reset counter, the hypervisor could deliver an interrupt to the OS when a reset occurs. Similarly, the OS could deliver a signal to registered applications which would either cancel ongoing transactions or perform other reset-related exception handling. This approach may prove ideal, because it removes the overhead associated with both system calls and hypercalls.

The current solution doesn't necessarily support nested virtualization. If the modified hypervisor is in fact a nested hypervisor, it could be reset along with all of it's guest VMs. Our solution requires lower-level hypervisors to inform higher-level operators of VM-resets. Because VM-resets are, by design, transparent to those being reset, this is inherently a problem.

7 Conclusion

Our approach can be used to provide atomicity for sensitive code across VM-resets whenever the hypervisor provides some form of hypercall interface to guest VMs and

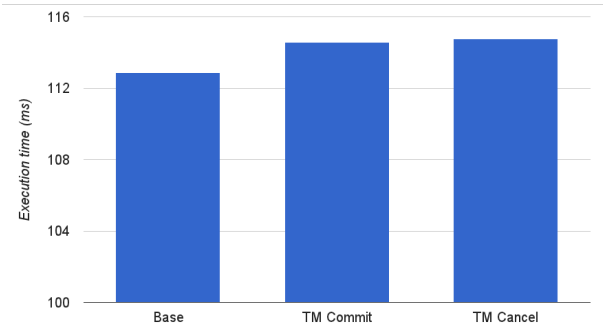

Figure 7: encryptSign runtime with TM commit and cancel

a transactional memory implementation is available. Our approach requires minimal modifications to the hypervisor, OS, and sensitive applications. This approach does add approximately 50% performance overhead to typical cryptographic operations, but can be used to restore security guarantees when random values and other one-time-use values are used in security operations such as encryption and digital signatures. This approach has no performance impact on unmodified applications on the guest VM or other parts of a software application that are not reset-sensitive. To our knowledge, this is currently the only solution that has been proposed to guarantee atomicity across a VM-reset.

8 Future Work

Future work on this topic will focus on reducing the performance overhead associated with guaranteeing reset-atomicity. If the alternative approach that uses interrupts and signals suggested above is used, all of the system call and hypercall overhead is eliminated (excluding a single system call required to register a reset-sensitive applications with the OS). This would leave only transaction-related overhead. TM is an active research area and hardware vendors promise support for TM in future CPU architectures [6][11][16][21], so TM-related overhead may be much more reasonable on future hardware.

Additionally, it would be valuable to demonstrate that real-world applications like Firefox and Apache can use this technique to fix known vulnerabilities. Such work could investigate both the performance impact of this solution as well as the complexity of the required software changes. For instance, we've discovered that GCC's TM implementation requires code to be carefully factored into revocable and non-revocable functions (they cannot be mixed when called from within an atomic transaction) and irrevocable functions must be individually marked under the current GCC TM specification.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[2] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 232–249, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "pseudo-random" number generation within cryptographic algorithms: The dds case. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 277–291, London, UK, UK, 1997. Springer-Verlag.

[4] Dave Boutcher. Software transactional memory in gcc. http://www-users.cs.umn.edu/ boutcher/stm/, January 2013.

[5] Bill Childers. Virtualization shootout: Vmware server vs. virutalbox vs. kvm. *Linux J.*, 2009(187), November 2009.

[6] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman. Asf: Amd64 extension for lock-free data structures and transactional memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 39–50, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[8] Pamela C. Durham. Virtual machine migration. US Patent 20 120 266 163 A1, October 2012.

[9] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[10] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, New York, NY, USA, 2012. ACM.

[11] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Michael Kozuch, M. Satyanarayanan. Thomas Bressoud, Michael Kozuch, M. Satyanarayanan, M. Satyanarayanan, Thomas Bressoud, Thomas Bressoud, Yan Ke, and Yan Ke. Efficient state transfer for internet suspend/resume. Technical report, Intel Research Laboratory at Pittburgh, 2002.

[13] H. Andrés Lagar-Cavilla, Joseph A. Whitney, Roy Bryant, Philip Patchin, Michael Brudno, Eyal de Lara, Stephen M. Rumble, M. Satyanarayanan, and Adin Scannell. Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst.*, 29(1):2:1–2:45, February 2011.

[14] Patrick Marlier. Brief transactional memory gcc tutorial. http://pmarlier.free.fr/gcc-tm-tut.html, May 2012.

[15] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006.

[16] James Reinders. Transactional synchronization in haswell. http://www.doc.ic.ac.uk/ phjk/GROW09/papers/03-Transactions-Schwindewolf.pdf, February 2012.

[17] Torvald Riegel. Transactional memory in gcc. http://gcc.gnu.org/wiki/TransactionalMemory, February 2012.

[18] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*. The Internet Society, 2010.

[19] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, and Luca Benini. Towards Transactional Memory Support for GCC. In *1st GCC Research Opportunities Workshop*, Paphos, Cyprus, January 2009.

[20] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[21] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 127–136, New York, NY, USA, 2012. ACM.

[22] Ming Zhao and Renato J. Figueiredo. Experimental study of virtual machine migration in support of reservation of cluster resources. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, VTDC '07, pages 5:1–5:8, New York, NY, USA, 2007. ACM.