Toward Dependency-Aware Live Virtual Machine Migration

Anthony Nocentino and Paul M. Ruth
Dept. of Computer and Information Science, University of Mississippi
University, MS, USA
aen@cs.olemiss.edu, ruth@cs.olemiss.edu

ABSTRACT

The most powerful characteristic of any machine virtualization technology is its ability to adapt to both its underlying infrastructure and the applications it supports. Possibly the most dynamic feature of machine virtualization is the ability to migrate live virtual machines between physical hosts in order to optimize performance or avoid catastrophic events. Unfortunately, the need for live migration increases during times when resources are most scarce. For example, load-balancing is only necessary when load is significantly unbalanced and impending downtime often causes many virtual machines to seek new hosts simultaneously. It is imperative that live migration mechanisms be as fast and efficient as possible in order for virtualization to provide dynamic load balancing, zero-downtime scheduled maintenance, and automatic failover during unscheduled downtime.

This paper proposes a novel dependency-aware approach to live virtual machine migration and presents the results of the initial investigation into its ability to reduce migration latency and overhead. The approach uses a tainting mechanism originally developed as an intrusion detection mechanism. Dependency information is used to distinguish processes that create direct or indirect external dependencies during live migration. It is shown that the live migration process can be significantly streamlined by selectively applying a more efficient protocol when migrating processes that do not create external dependencies during migration.

Categories and Subject Descriptors

C.4 [**Computer-Communications Networks**]: Miscellaneous

General Terms

Performance

Keywords

Virtualization, Virtual Network, Machine Migration

VTDC'09, June 15, 2009, Barcelona, Spain.

1. INTRODUCTION

Virtual machines are becoming increasingly prolific in modern computer systems. We have seen virtual machines deployed in modern data centers, as computational clusters, and even on individual desktop computers. The primary reason driving the popularity of virtual machines is their ability to dynamically adapt to the changing needs of administrators, applications, and users [10, 11].

Possibly the most powerful feature of machine virtualization is its ability to migrate virtual machines from one host to another. Live migration [3] enhances this power by allowing virtual machines to migrate without being paused or losing network connectivity. This ability is fundamental to machine virtualization's claims of dynamic load balancing, zero-downtime scheduled maintenance, and automatic failover during unscheduled downtime.

Many popular virtualization platforms, including Xen [1] and VMware [12], provide live migration facilities. Live migration works well on these platforms and is used for many research and industrial purposes. On one hand, virtual machine migration is heavy-weight and inefficient. Light-weight process virtualization [8], on the other hand, provides a more efficient mechanism for migration but is limited in functionality.

The ideal migration facility would have the following characteristics:

1. Interaction would be through a traditional execution environment that supports legacy applications.
2. Migration latency (i.e., the time elapsed from the initiation of migration to the arrival at the destination) would be minimized.
3. Resource overhead (primarily caused by network traffic) of the migration process would be minimized.

This paper presents an initial investigation toward dependency aware live virtual machine migration. This technique incorporates some of the performance benefits of process migration into fully realized virtual machines. The primary contributions of this paper are: 1) The novel use of process tainting [6] to identify external dependencies created by processes; 2) the collection and use of dependency information within Xen's live migration facility; and 3) results measuring reduction in network traffic that can make dependency-aware live virtual machine migration viable.

Dependency-aware live virtual machine migration can reduce the overhead of live migration to that of non-live migration. Further, the largest benefit of dependency-aware

virtual machine migration can be seen while migrating processes that demonstrate characteristics that are the most problematic for existing live migration mechanisms. For example, the MMuncher program used to demonstrate the worst case scenario in Xen's migration evaluation [3] is our best case scenario.

Dependency-aware virtual machine migration does not aim to provide better performance than process migration, but instead works from the assumption that users want a fully featured execution environment with fully transparent migration capabilities. This assumes that users are willing to accept some performance overhead. The assumption that users prefer full functionality to cutting-edge performance is evidenced by the far-reaching adoption of machine virtualization despite it being heavy-weight.

The remainder of this paper is organized as follows: Section 2 discusses current live virtual machine migration mechanisms. Section 3 presents our dependency-aware migration mechanism and discusses its benefits. Section 4 discusses current implementation efforts while Section 5 shares and analyzes performance results gathered using the implementation. Finally, Sections 6, 7, and 8 discuss related and future work and conclude the paper.

2. LIVE VIRTUAL MACHINE MIGRATION

Many traditional virtualization platforms, including VMware and Xen, can migrate virtual machines from one host to another. Fundamentally, virtual machine migration is achieved by transferring the virtual machine's state from a *source host* to a *destination host*. The simplest techniques pause the virtual machine by first saving its state to a file, then transferring the file from the source host to the destination, and finally resuming the virtual machine from the saved state. These non-live migration techniques require the virtual machine to be paused for the duration of the migration. Services provided by the virtual machine will not be accessible during the migration, and open network connections may timeout and disconnect.

More advanced live migration mechanisms do not require a noticeable pause of the virtual machine's execution. Instead they iteratively transfer the virtual machine's state to the destination host while allowing the virtual machine to continue its execution on the source host. The most significant portion of a virtual machine's state is its memory, and the iterative copying of memory composes most of the overhead of live migration.

Virtual machines do not use physical memory directly. Instead they use abstracted *pseudo-physical memory*. Further, a virtual machine's pseudo-physical memory *frames* are memory *pages* from the perspective of the host and are referred to as *pseudo-physical memory pages*, or simply *pages*, throughout this paper.

In order for a virtual machine to be live during migration, its pseudo-physical memory pages must remain accessible. The iterative transfer of memory state allows for the use of memory during migration as long as any pseudo-physical memory that changes (i.e. is written to) is marked as *dirty* and will be re-transfered during a subsequent iteration. Live migration continues iteratively transferring state data until the amount of state is less than a defined tolerance. During the final iteration the virtual machine is paused, transfered, and resumed using a technique similar to non-live migration. However, owing to the reduced amount of state data the migration downtime can be as little as tens of milliseconds [3]. The minimal downtime of live virtual machine migration creates the illusion that the virtual machine migrated without interruption.

The first two iterations of a typical live migration mechanism are shown in Figure 1. In the example, the first iteration must transfer all pages of the virtual machine's memory. However, only half of the memory has changed and is marked *dirty* during the first iteration. This allows fewer pages to be transfered during the second iteration.

Iterative virtual machine migration techniques are extremely effective. However, they are often criticized as being heavy-weight when compared to process migration. This critique is amplified when one considers the varied reasons for migration. Many times live migration is initiated by an event that requires the migration to finish quickly. Further, an event that initiates a migration will often initiate several migrations simultaneously. For example, automated failover during a catastrophic event may cause all virtual machines residing on a single host (or in a single rack) to migrate at the same time. The dynamic load-balancing policy of a computational cluster may cause many virtual machines to migrate simultaneously. Further, migration is often performed during times when resources are scarce. In fact, high resource demand can in itself be the cause for migration. In these situations, high overhead of live migration could cause more problems than it solves, leading to even more contention for resources.

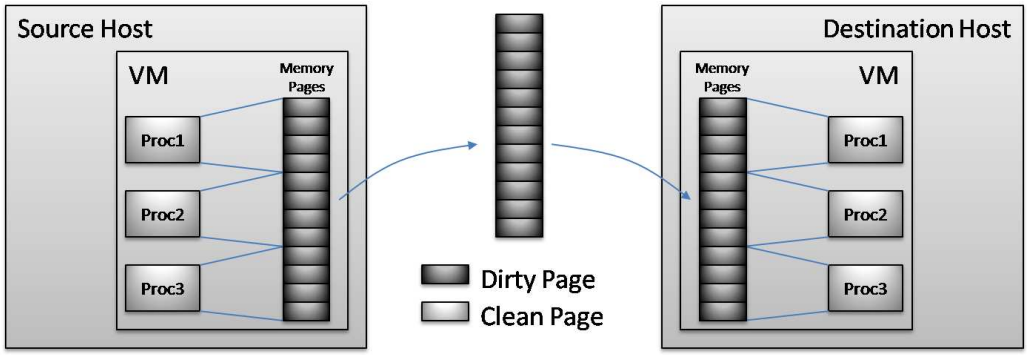
It is clear that a migration technique that allows for live migration is preferable to one that does not. However, live migration's use of iterative copy increases the latency of migration and is the source of significant network overhead. The most efficient live migration mechanisms will copy the memory as few times as possible.

Additionally, iterative copy has specific difficulty migrating applications that repeatedly dirty many memory pages during each iteration. These applications do not allow for a sufficiently small amount of state to be copied during the pause/restore phase. The more memory that is dirtied during an iteration the more state data must be transfered.

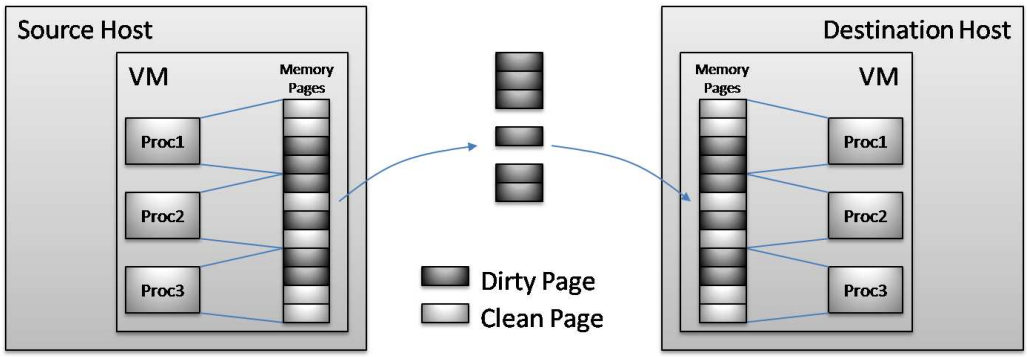
3. DEPENDENCY-AWARE LIVE VIRTUAL MACHINE MIGRATION

Dependency-aware migration grew from the observation that a virtual machine that does not form dependencies through interacting with external objects (i.e., other virtual or physical machines, or devices such as a hard-disks and consoles) can be migrated *live* without being *alive* during the migration. In other words, a virtual machine that does not interact with the outside world can achieve the illusion of live migration while being paused, migrated, and resumed.

To understand this claim we must start with the questions: "*Why do we need the virtual machine to be live during migration?*" and, "*Why does iterative migration re-transfer dirty state after each iteration?*" It is desirable for virtual machines to be live during migration because external objects may need to interact with the virtual machine during migration. We need to re-transfer dirty state because interactions cause external objects to know that a virtual machine has progressed to the point at which the interaction occurred. The state that was originally transfered to the destination



(a) First iteration of an iterative live migration.



(b) Second iteration of an iterative live migration.

Figure 1: First two iterations of an iterative live migration.

has become inconsistent with respect to the external object and cannot be used.

Alternatively, suppose we have a virtual machine that does not interact with the outside world or at least will not interact with the outside world for the duration of a migration. For example, the machine may be executing a CPU intensive physics simulation or a temporarily idle web server. If that machine were to be migrated using iterative live migration, the entire state of the machine would be transfered on the first iteration. When the first iteration finishes there will be dirty state (i.e. dirty memory pages). However, no external object would know that the virtual machine had progressed past the point that the state was dirtied. In other words, there will be no external machine, process, file, human, or other object that will *depend* on the virtual machine maintaining its most updated state. Also, observe that the state snapshot stored destination host is a consistent, but slightly old, copy of the virtual machine. Since there is no dependency on the current dirty state, that state can be abandoned and the destination's old snapshot can be safely resumed.

The previous example is similar to non-live migration in which a virtual machine is paused, a snapshot of its state is transfered, and it is resumed on the destination host. However, the difference is that it maintains a running copy of the virtual machine as a contingency against any new dependencies that may be created during the migration. If no dependencies are formed, the *contingent virtual machine* is not needed and the originally transfered state can be resumed.

This observation applies not only to a virtual machines but to the individual processes within them. If we can identify processes that do not create new dependencies during migration, we can selectively migrate dependent and independent processes. Fundamentally, dependency-aware migration aims to identify dependencies between processes and objects outside of the virtual machine. Processes that do not create new dependencies during a migration can be migrated quickly, while processes that create new dependencies must be subject to the full overhead of the live virtual machine migration mechanism.

3.1 Process Dependencies

Migration can be implemented at levels other than that of the virtual machine. Within a computer, virtual or physical, the basic abstraction representing an executing program is the process. Each process has its own memory address space usually composed of pages logically storing data that the process is using. A process is itself a good candidate to be used as a container for migration. In fact, several projects have succeeded in enabling process migration [8]. As discussed in previous sections of this paper, process virtualization and migration can be more efficient, but it is limited in functionality and portability when compared with full machine virtualization.

The bulk of a process' state is composed of the pages in its address space. A process can be paused, transfered, and resumed, like a virtual machine, assuming that the destination operating system has the same network settings, filesystem, processes, and kernel data structures (or if the process does

not utilize any of these entities). Full virtual machine migration ensures these requirements are met by migrating the process and its operating system together.

The idea of employing contingent virtual machines can be applied at a finer granularity than processes. During virtual machine migration, a process' state can be transfered to the destination host along with the rest of the virtual machine. Meanwhile, a *contingent process* continues to execute on the virtual machine on the source host. The destination host's copy of the process' state will remain consistent until the contingent process creates a dependency with an external object (for example, a file read or write). The dirtied state of the process can be abandoned, and the slightly old but still consistent state that is already on the destination can be resumed if no dependencies are created between the contingent process and an external object during the remainder of the migration.

3.2 Process Tainting

In order to selectively migrate only memory pages belonging to contingent processes that have created external dependencies we must be able to identify dependencies and assign these dependency to processes and their memory pages. The security community has developed provenance-award intrusion detection mechanisms that can be adapted for this purpose [6, 5]. Dependency-aware migration utilizes an intrusion detection technique, process tainting, which is based on tracking information flow through a system. These systems track *taint* from a potential break-in point to all processes that may have been effected by the break-in. In dependency-aware migration these potential break-in points are not considered malicious. Instead they are the initial source of the external process dependencies.

Taint-based intrusion detection systems assume that external interactions may results in malicious external objects, such as worms, injecting malicious code into a system. After an external interaction occurs, the process that participated in the interaction is considered *tainted*. After being tainted a process may interact with other processes either directly, through process creating or signaling, or indirectly through a file, network socket, or other form of IPC [6]. When a tainted process interacts with another process the taint is passed to the untainted process. As processes interact with each other, taint is diffused creating the set of all processes that may have been effected by a malicious external interaction.

Processes in dependency-aware migration can be directly or indirectly dependent on an external interaction. Further, the diffusion of taint via process interactions is required, although in the reverse direction. For example, if process A and process B interact they become dependent to each other. If at any time after their interaction, either process A or process B interact with an external entity then both A and B become dependent on the external event. More intuitively, if process A interacts with an external object, that object clearly knows that process A has progressed to the point of their interaction. Less obviously, process A knows that process B has progressed to the point of their interaction. Therefore, process B is now indirectly dependent on the external event in which process A participated. More complex process interactions exist and will be the subject of future study.

3.3 Proposed Migration Mechanism

During its normal operation, dependency-aware migration does not effect the virtual machines it supports. Virtual machines function exactly as they would without dependency-aware migration. When a migration is initiated, the process tainting mechanism within the virtual machine is enabled. At that point, processes within a virtual machine that interact with external objects create dependencies that *taint* the process. Following the process taint dispersion mechanism, processes that interact with each other are indirectly marked as tainted as well.

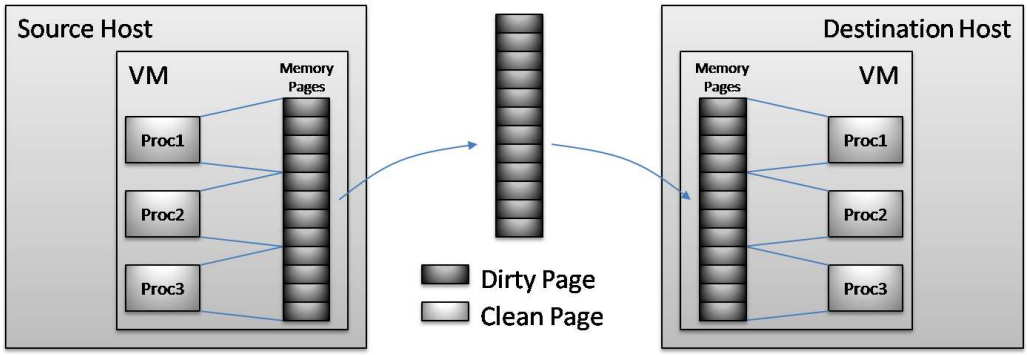
When live migration is initiated the virtual machine monitor begins copying the virtual machine's pseudo-physical memory pages from the source host to the destination. The first iteration of the live migration requires transferring all pseudo-physical memory pages, as shown in Figure 2(a). All subsequent iterations must only transfer the memory pages that are both *dirty* and belong to a process that has created an external dependency since it was last transfered. When processes create dependencies their pages are marked tainted. So, each iteration of a live migration must copy only pages that are both *dirty* and *tainted*, as shown in Figure 2(b). When a page is transfered it is reset to both clean and untainted. This results in a reduction in the number of pages transfered that is equal to the number of dirty pages that are not tainted.

When migration is complete the entire virtual machine, including all of its processes, will be executing on the destination host. All processes will be guaranteed to be in a consistent state with respect to external entities. However, the state of some processes will have been updated since the initial snapshot that was transfered in the first iteration, while other processes will be resumed from the state transfered during earlier iterations. These slightly old snapshots are consistent with respect to external entities because they are guaranteed to not have interacted, directly or indirectly, with any external entities.

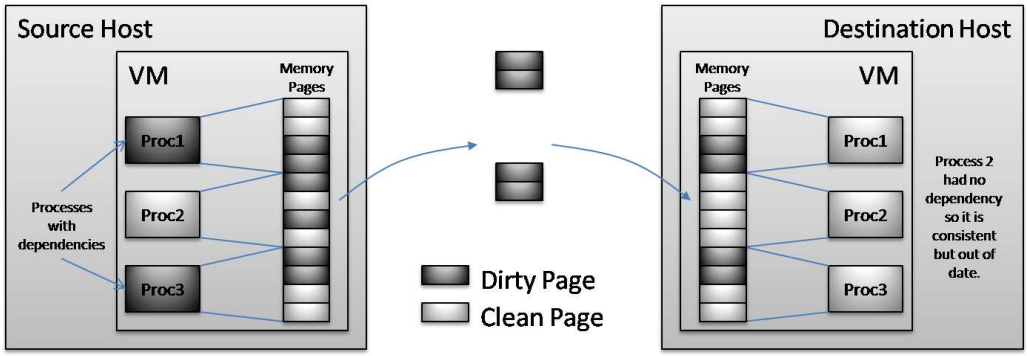
The primary benefit of dependency-aware virtual machine migration is to the reduction of migration latency, downtime, and the overhead of migration. Virtual machines supporting CPU dependent applications will benefit the most. It may seem that very few applications exhibit these characteristics. However, dependency-aware migration does not prohibit applications from creating external dependencies. Instead, it only requires applications to not create external dependencies for the relatively short duration of a single migration iteration. Many distributed computational applications oscillate between phases of computation and phases of communication. Live migration during a computation phase could be performed in as little as a single iteration. Grid and cluster platforms may receive extreme benefits from dependency-aware migration. As virtual machines are increasingly deployed on grid and cluster computers [4, 7, 2], virtual machine migration will gain in importance.

4. IMPLEMENTATION

This section describes the initial implementation of the dependency aware virtual machine migration. The goal of the initial implementation was not to provide fully featured dependency-aware migration. Instead, the intended goal was to achieve enough functionality to collect data that demon-



(a) First iteration of a dependency-aware live migration.



(b) Second iteration of a dependency-aware live migration.

Figure 2: First two iterations of a dependency-aware live migration.

strates the potential reduction in migration overhead a fully featured system could provide. Section 4.1 describes the mechanism to diffuse taint to dependent processes. Section 4.2 describes how the implementation manages the migration process to collect and report dependency data during a live migration.

4.1 System Call Interception and Dependency Diffusion

Most operating systems, including Linux, provide access to its underlying hardware, memory, disk, and other devices through an interface composed of *system calls*. The migration facility needs to intercept these system calls to build dependency data and tag processes as dependent.

The Linux Security Modules (LSM) [13] are used to intercept the necessary system calls. The LSM interface mediates access to internal kernel objects, placing hooks in the kernel code ahead of the access. The intent of LSM is to be a security framework, however its distinct ability to mediate system calls has made it an obvious choice for prototyping systems that may ultimately require large amounts of development. The functionality provided by LSM allows the system to easily position itself between the virtual machine's user space and hardware.

LSM is installed on the virtual machine and is used to capture system calls which create or diffuse taint (i.e. external dependencies) within the virtual machine's process hierarchy. Using LSM allows the virtual machine to easily monitor access to resources including sockets, files, I-nodes and other I/O devices. Implementation of the dependency diffusion model required the use of LSM operations that will capture external input and output from our system, including process creation, IPC, file access, and socket access. When processes interact with each other, LSM captures the interaction and creates dependencies based on the type of interaction and the participating processes.

4.2 Migration Facility

The prototype implementation is a modification of the open source Xen virtual machine platform. Xen uses a managed migration model to migrate virtual machines between physical hosts. A managed migration is performed by the underlying hypervisor and tools under its control. By default, Xen live migration iteratively copies a virtual machine's pseudo-physical memory pages from a source host to a destination host. The hypervisor sits outside of the virtual machine and does not have knowledge of the the virtual machine's traditional high level abstractions including processes and files. It only knows about low level units such as disk blocks and memory pages. This gap in knowledge has been referred to as virtualization's *semantic gap*[9].

To bridge this gap, a dependency-aware migration facility must provide knowledge of the virtual machine's internal operating system to the hypervisor. The initial implementation uses the Linux operating system within its virtual machines. The migration facility has knowledge of the standard location of the Linux page descriptors. Using its knowledge of the virtual machine's internals, the migration facility is able to access information stored in the memory page descriptors and process control blocks [9].

When live migration is initialized, the virtual machine's

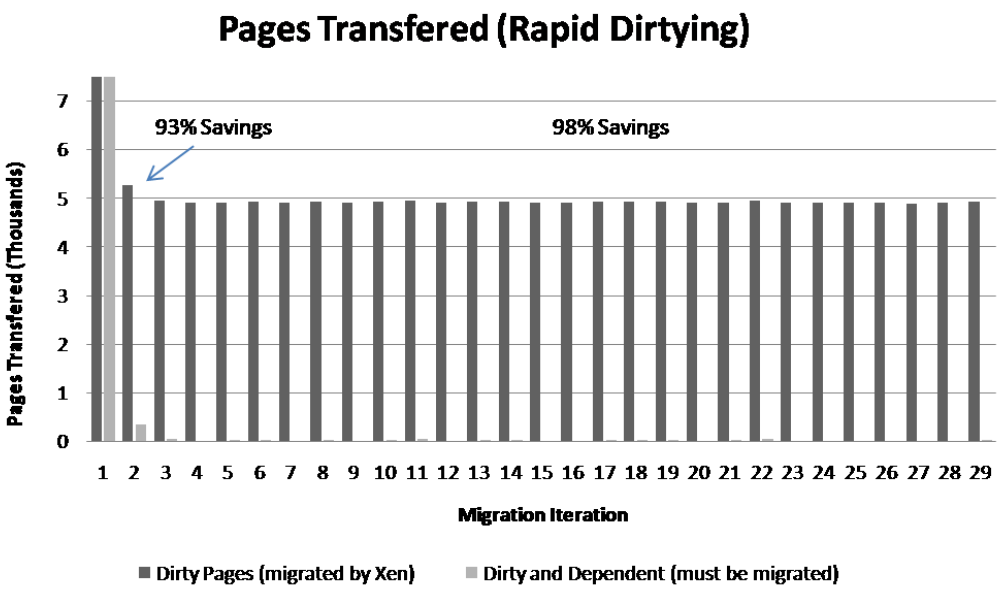


Figure 3: Number of memory pages transfered during iterative migration and the potential savings of dependency-aware migration during a CPU dependent computation.

page descriptors and process control blocks are accessed from the external migration facility. Information regarding dirtiness and taint (i.e. external dependencies) is used to determine if a page should be transfered during the current iteration.

Specific knowledge of the type of guest operating system is required. However, there is no reason why the ideas behind the system could not be applied to other guest operating systems.

5. EXPERIMENTAL RESULTS

The system was evaluated using two experiments. In both experiments a virtual machine was instantiated on one of two hosts, an application was executed within the virtual machine, and dependency data was collected during the migration. The migration of the virtual machine is not modified by the current implementation. However, the number of pages transfered (i.e. the *dirty* pages) during each iteration of the standard Xen migration process is recorded along with the number of pages that would be transferred using the dependency-aware migration model (i.e. the pages that are both *dirty* and *tainted*). It is shown that there can be a significant number of pages that are transfered by the standard mechanism that do not need to be transferred under the dependency-aware model.

The development and test environment consists of two Dell PowerEdge 1900 servers, each with two quad core Intel Xeon series 5355 2.66 GHz processors, 4GB of primary memory and a system bus speed of 1333 MHz. Both servers are configured with Xen 3.3.0 and use 32 bit Ubuntu 8.0.4 LTS running an SMP kernel (2.6.18.8) for the host operating systems. The guest operating system is paravirtualized 32 bit Ubuntu 8.0.4 LTS with Linux kernel 2.6.18.8, the virtual machine has 2GB of main memory and 10GB hard disk. The servers are connected to a private 1GB Ethernet network.

5.1 CPU Dependent Application

The first experiment used a small C program that allocates a large amount of memory (100MB) and repeatedly iterated through the memory writing data. This is the worst case test on standard live virtual machine migration. The program dirties pages at such a rapid rate that iterative live migration can never reduce the amount of dirty memory below the level necessary to commit a migration. In Xen, a live migration is limited to 30 iterations after which the migration is committed regardless of the number of dirty pages remaining. If too many dirty pages remain, the final iteration will not be transparent to the application. In extreme cases, the final iteration could take long enough to disrupt network connections.

The chart in Figure 3 shows the result of migrating Xen's worst case application. Xen very quickly stabilizes at just under 5000 dirty pages per iteration. While dependency-aware migration produces between 13 and 60 pages after only two iterations. It is common to see the final iteration occur once the number of dirty pages falls below 100, although the actual number is dependent on network performance. In this, case dependency-aware migration would have committed on the third iteration and would have have transmitted 93% and 99% fewer pages on the second and third iterations respectively.

5.2 Network Dependent Application

For our second case we tested an application that downloads a 4 GB file over HTTP. This case has a persistent external dependency and should not perform any better using dependancy-aware migration. Figure 4 shows that this process dirties pages at a very rapid rate and nearly all pages must be transfered during each iteration. The persistent dependency does not allow any significant performance gain.

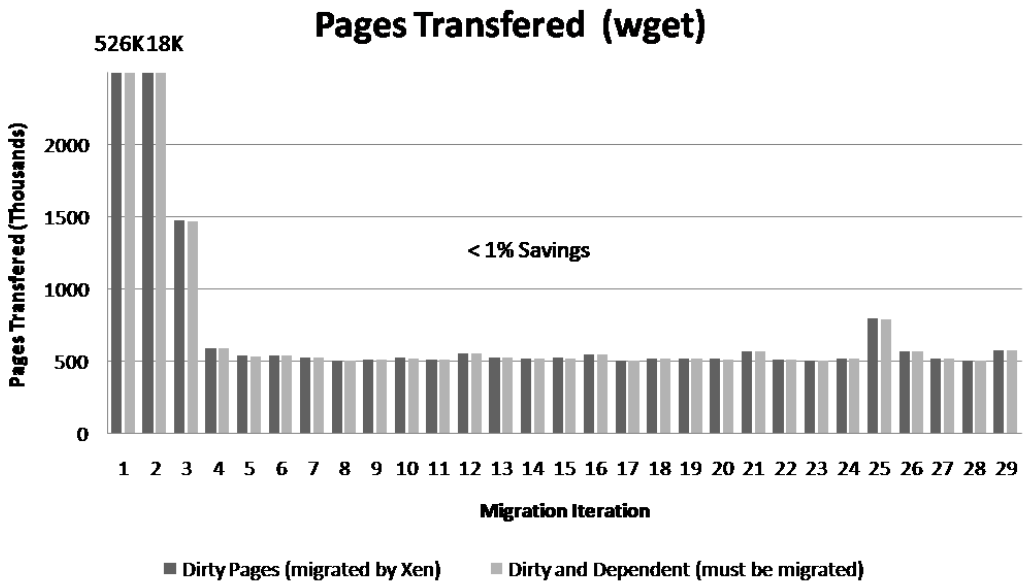


Figure 4: Number of memory pages transfered during iterative migration and the potential savings of dependency-aware migration during an HTTP file transfer.

6. RELATED WORK

Dependency-aware virtual machine migration has its roots in security and process tainting and coloring [6]. The basis of our dependency-aware migration model is process coloring's diffusion model. A significant amount of work has been done involving process tainting in the field of security. However, to the best of our knowledge, this is among the first projects outside the field of computer security that uses taint.

Both Xen and VMware have full featured, live migration implementations, each of these employs a managed, heavy-weight migration scheme. Additionally, process migration aims to provide a lightweight migration scheme, but has its pitfalls in managing external dependencies [8]. However, this comes at the cost of handling special situations, for example, open network connections, and files. Current process migration techniques employ restrictions on which processes can be migrated or utilize proxies that forward information to the process' destination.

7. FUTURE WORK

The primary focus of future work on this project will be into formalizing and modeling more complex dependency relationships between processes. This model will enable us to prove the consistency of virtual machine processes and will provide us with the confidence to continue this line of research. Specifically, through modeling indirect dependencies between processes and external objects we hope to further decrease the overhead of virtual machine migration.

Additionally, we intend to complete the dependency-aware migration functionality. We have made significant progress in the implementation of the process tainting mechanism into Xen's process migration facility; however, the complete selective migration facility remains to be completed.

8. CONCLUSION

We have presented a novel model for quick live migration of full virtual machines using dependency-aware selective migration of processes. As virtual machines increase in popularity and more autonomic migration capabilities are included into environments of virtual machines, the need for more efficient live migration mechanisms will similarly increase. The need for increased migration is even more apparent when we consider that migration often occurs at the worst time possible. Particularly, migrations tend to occur during time of increased contention for resources or impending catastrophic events. We have shown that the potential performance gains using dependency-aware migration can be extreme and are particularly applicable to applications that are not handled well by current migration mechanisms.

9. REFERENCES

[1] 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 *ACM SOSP*, 2003.

[2] J. Chase, D. Irwin, L. Grit, J. Moore, and S. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *IEEE HPDC*, 2003.

[3] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of Virtual Machines. In *USENIX NSDI*, 2005.

[4] R. Figueiredo, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. In *IEEE ICDCS*, 2003.

[5] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[6] Xuxian Jiang, Florian Buchholz, AAron Walters, Dongyan Xu, Yi-Min Wang, and Eugene H. Spafford. Tracing worm break-in and contaminations via process coloring: A provenance-preserving approach. *IEEE Transactions on Parallel and Distributed Systems*, 19(7), July 2008.

[7] Katarzyna Keahey, Karl Doering, and Ian T. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *GRID*, pages 34–42, 2004.

[8] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments, 2002.

[9] Bryan D. Payne, Martim D. Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 2007 Annual Computer Security Applications Conference*, 2007.

[10] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual Distributed Environments in a Shared Infrastructure. *IEEE Computer*, 38(5):63–69, May 2005.

[11] Paul Ruth, Junghwan Rhee, Dongyan Xu, Rick Kennell, and Sebastien Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. In *Proceedings of The 3rd IEEE International Conference on Autonomic Computing*, June 2006.

[12] VMware. http://www.vmware.com.

[13] Chris Wright, Crispin Cowan, and James Morris. Linux security modules: General security support for the linux kernel. In *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.