SLYK: A Transparent Fault-Tolerant Migration Platform

Jasper Lin, Jennifer Shu, Olivier Koch, Shuchyng You
{jasperln, jshu, olivierk, yoshu117}@mit.edu

Abstract

The recent trend towards mobile computing has introduced new challenges such as migrating a user's computing environment as he moves from location to location. Although laptops offer a great deal of mobility, they still suffer from traditional drawbacks, such as having weak computing power compared to desktops and being relatively expensive and encumbering. In the past few years, the concept of a virtual environment that can be suspended at one place and resumed at another has started to emerge, opening the door to true mobility.

We propose a virtual machine-based migration platform that preserves active network connections across machine migrations. To make our system fully deployable, we require neither cooperation from the outside world nor any modification to the host or guest operating system. The platform provides machine and network transparency as well as fault tolerance and data integrity.

1 Introduction

In today's computing environment, it is common for one user to encounter several different computers in the course of a day. Computers are increasingly being viewed as public utilities that are as ubiquitous as electricity and water. As a result, users no longer need to value computers as an expensive resource; instead, they can place greater worth on the state, including personal data and open applications, stored on these computers.

A mobile working environment would be useful to virtually anyone who uses a computer. For instance, a student working on a desktop in his lab could transfer his state to his laptop at home, and be able to switch back and forth every day. If he were in the middle of a long simulation, for example, he could resume it on different computers without having to start over or wait in lab until it finished. Similarly, a businessman could travel across the United States and recover his latest work on any computer with Internet access.

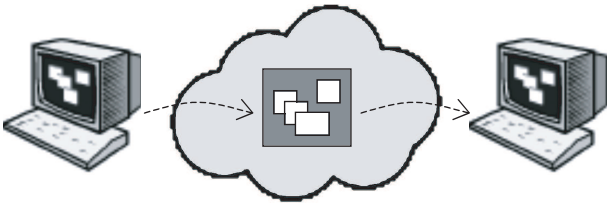

Figure 1: SLYK facilitates the migration of user state between network-connected machines.

This novel view of computing, however, opens up a new batch of interesting challenges that need to be addressed before such a system can be successfully deployed. First, for maximum deployability, no cooperation from the rest of the network should be required. In other words, if a user is migrating from one machine to another, the rest of the network should not be aware of the change, and the user's active network connections should not be broken. We refer to this challenge as *network transparency*. Second, in order to offer the most flexibility, the migration platform needs to be hardware and operating system independent. One promising approach is to use virtual machines (VMs) which, by nature, do not depend on the underlying operating system and therefore allow true *machine transparency* [8]. Third, our system needs to provide a *fault-tolerant* approach for data storage. If a machine running the migration platform suffers a hard disk (HD) failure, for example, then other machines should be able to resume without loss of data.

In this paper, we attempt to address the above challenges, while focusing on the problem of migrating user state as users traverse machines. We propose SLYK[1], a virtual machine-based migration platform that preserves active connections across machine migrations and offers a high level of fault-tolerance. Since a VM simulates a complete architecture, users

[1]SLYK, pronounced "Slick", is taken from the authors' last names: Shu, Lin, You, and Koch.

are permitted to run any operating system and application compatible with the emulated architecture. The state of any virtual system running on SLYK can be packaged and sent over a network to be resumed by any other machine running SLYK, as shown in Figure 1.

2 Related Work

Previous work on using VMs to migrate state focuses mainly on optimizing the performance of emulation and the speed of migration [15]. Although performance is important for the mainstream adoption of VMs, there are other important factors such as fault-tolerance and transparent operation with the outside world.

Internet Suspend and Resume (ISR) [9] presents a straightforward implementation of a VM migration infrastructure. Upon suspend, the state of the VM is stored on a remote NFS server. When resumed, the state of the VM is copied from NFS onto the target machine and the VM is started.

Optimizing the Migration of Virtual Computers describes several optimizations to speed migration time [15]. Their goal is to make it practical to migrate state between home and work computers over a 384kbps link. During the process of migration, virtual HD blocks are left on the source machine to be requested as needed by the target machine.

Both of these systems are based on VMWare [13], so they only work on the x86 platform. Additionally, these projects suffer from two other drawbacks. First, all active network connections are lost during migration. Applications that depend on these connections need to be reset on the target machine. Second, HD blocks which have not been requested and cached locally may become inaccessible when their host machines go down.

Mobile IP [14] provides mobility by always routing packets first to a static home host then to the mobile host, which works when a static host is always available and not separated by the network. However, failure of this home host results in loss of all active mobile connections. SLYK uses a Mobile IP-like infrastructure to migrate active connections, but the home host can be dynamically specified.

There have been several proposals to migrate state at a finer granularity than full system migration [3, 18, 21]. These systems exploit specific knowledge about the state or execution environment to ship the minimum amount of data needed for seamless transition. In contrast, VM approaches involve potentially needing to send much more state than needed. However, the general approach adopted by VM migration platforms allows the migration of many more operating systems and applications without any modification. Furthermore, several optimizations can be performed to reduce the inherent overhead of migration via VMs [13, 15].

3 Challenges

We face many difficulties in designing a migration platform that is transparent, fault-tolerant, and efficient. In order for a migrated machine to function as if it were still operating on the original host, we need to transfer a large amount of state, including that of the HD, RAM, and CPU. Unfortunately, the majority of state is kept on the HD, and it is both inefficient and unnecessary to transfer the entire HD during migration. However, since we would like to present a consistent snapshot of a user's personal state across different machines, we need to optimize HD migration while transferring enough state for full recovery from possible system failures. Instead of stalling migration until the entire HD is transferred, our system fetches data blocks (sectors) on demand, thereby increasing the speed of migration.

Clearly, using the host machine's HD to store the VM state is hazardous, because the failure of the host machine might render some crucial data inaccessible to other host machines. We aim to address this problem by using a Distributed Hash Table (DHT) store acting as a virtual HD to store sectors of the HD image. Using a DHT requires a communication scheme to load and fetch data sectors from the DHT.

One of the many goals we aim for in SLYK is network transparency; as users migrate, their active connections should stay intact. For example, network transparency would allow a SLYK user to begin downloading a file on one SLYK-enabled machine and continue downloading the same file when he migrates to another machine. To achieve this goal, we need a packet forwarding mechanism to allow packets to be re-routed when a SLYK user moves between hosts with different IP addresses.

To make SLYK as deployable as possible, we don't want to require any modifications of applications or the guest or host operating systems. Ideally, SLYK would run as an application on the host machine, without requiring administrator privileges. Transferring data from one host O/S to another some-

times requires dealing with different endian formats and data representations. SLYK needs to overcome these differences in order to be portable to all of the commonly-used platforms, such as Linux, OS X, and Windows.

In the past, a lot of work has been done on migrating state without keeping the active connections alive [9], or optimizing performance while sacrificing fault-tolerance. SLYK is different from previous projects in that it incorporates machine and network transparency, fault-tolerance, and deployability into one migration platform.

4 System Overview

SLYK is a migration platform that allows state to be transferred across several different architectures, while maintaining network transparency and providing fault-tolerant data storage. Starting with an open source emulator, we have added the ability to transfer and resume a memory image, forward packets to handle active connection migration, and access sectors from a DHT store for the virtual HD.

4.1 Machine Emulation

SLYK is built on top of QEMU [2], a highly portable open source emulator that runs on both the x86 and PowerPC architectures. Utilizing dynamic translation, QEMU is a much faster processor emulator than Bochs [10]. QEMU also allows network communication without needing superuser privileges.

4.2 Remote Communication

Machines running SLYK communicate with each other using XmlRpc++ [12], an implementation of the XML-RPC standard. We chose to use XML-RPC as our communication protocol because it is an industry standard and is portable to many different platforms. We considered several implementations of XML-RPC. XmlRpc++ proved to be the most portable in that it builds on all three platforms we tested (Linux, OS X, and Windows) without requiring external libraries. SLYK also uses XmlRpc++ to store and retrieve data from the DHT.

RPC communication is exchanged between machines via a dedicated SLYK port[2]. Both ends of the communication send the payload (body of the request and response) over HTTP in XML format.

[2]Currently, the SLYK RPC port is 5544. The SLYK bulk transfer port is 5545.

4.3 Reliable Storage

To provide reliable storage, we chose to replicate the HD image on a DHT. The specific DHT that we use is OpenDHT [19], which runs on PlanetLab [1] with over 200 nodes available for answering put and get requests. OpenDHT is cross-platform, supports communication through XML-RPC, and provides reliable storage by replicating segments across its network of nodes. DHash [5], another DHT implementation, also could have provided the fault-tolerance that we desired. However, accessing services on OpenDHT does not require that the client machine be Unix-based. SLYK flushes and fetches data sectors on demand from OpenDHT.

4.4 Migration

State transfer amounts to serializing QEMU's RAM, CPU, and timer representations, sending it over the network, unserializing the image on the other end, and bootstrapping SLYK with this state. In addition, SLYK also sends along an array of hashes that is needed for fetching data sectors on demand from OpenDHT. These are the steps that SLYK follows when transferring ownership to the new machine:

1. The source machine sends an RPC to the destination machine to initiate the migration.

2. Upon receiving this RPC, the destination machine opens up a port to receive the bulk data transfer from the source machine.

3. The destination machine responds with the port number when it is ready to receive. Then the source machine begins transferring the serialized state and hash array.

4. When the state transfer is done, the source machine closes the TCP/IP port which indicates to the destination machine that it may load this state and resume execution.

5. The source machine then continues to flush any remaining dirty sectors it may have cached to OpenDHT.

6. The virtual system on the new machine can continue execution while this flushing takes place but blocks if it tries to demand-page a sector

from OpenDHT that hasn't yet been flushed, and resumes when the sector contents are finally flushed.

5 Virtual HD Store

Using the local HD of the SLYK machine clearly makes the system error-prone. If one of the SLYK machines were to go down, the data stored on its HD would be lost until the machine has recovered. This loss of data could prevent the user from resuming his activities on a new machine. In addition, transferring the content of a full HD over the network would make the migration very slow. Therefore, we provide a virtual HD that is accessible by all SLYK machines at all times.

From the point of view of a SLYK machine, disk I/O happens as if the local HD were being used, thereby making the use of the virtual HD fully transparent. The read and write operations to the local HD are simply intercepted and converted into read and write operations to the virtual HD. To keep migration time within a reasonable limit, only the virtual system's memory is transferred over the network during migration. Virtual HD blocks are demand-paged from OpenDHT and cached after the VM has already started on the new machine. The following sections describe how SLYK disk images are formatted, how SLYK communicates with OpenDHT, and how the cache is used.

5.1 DHT

The virtual HD is stored on OpenDHT. Since the virtual system runs only on one computer at a time, no complex coherency protocol between SLYK and the OpenDHT is needed. The first time a block is fetched, SLYK caches it in its local store. On writes, SLYK marks the block as dirty and eventually flushes the block out to OpenDHT.

OpenDHT offers a very transparent way to store and access <key,value> pairs. There is a 1024-byte limit on values in OpenDHT; therefore, data must be broken into blocks before being stored. Since QEMU's data storage is based on 512-byte disk sectors, we use these sectors as values in OpenDHT. It is important to note that data cannot be changed or removed once it has been stored on OpenDHT. Instead, a timeout mechanism makes the data obsolete on OpenDHT after a certain amount of time (up to one week). The timeout value is specified by the user and should be chosen carefully. Using a small timeout may result in loss of data, whereas using a large value may conflict with PlanetLab requirements limiting the amount of storage used by each client. However, the operability of SLYK is not impacted by this choice as long as a reasonable value is used. Our implementation uses a timeout of 30 minutes.

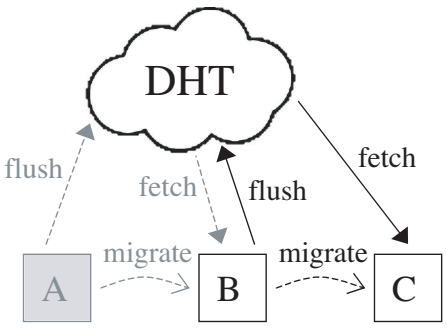

Figure 2: For fault-tolerance and fast migration, SLYK stores virtual HD blocks on an external OpenDHT store. HD blocks are paged in on demand and cached locally. Dirty blocks are periodically written out to keep OpenDHT updated. After migration, all dirty blocks are flushed to OpenDHT.

We store the hash of a sector's contents as the key and the sector's actual contents as the value in OpenDHT, using Secure Hash Algorithm 1 (SHA1) [6] for the key computation. It takes as input a message of less than 2^{64} bits and produces a 20-byte (160-bit) message digest. In the unlikely case of hash collisions, OpenDHT would return a list of values corresponding to the same hash because it does not overwrite values that have the same keys.

Data on OpenDHT should be kept as updated as possible; otherwise, flushing dirty blocks out to OpenDHT may become part of the critical path in migration. SLYK should not be too eager to flush blocks either, since batching writes to the same block into a single write to the DHT saves bandwidth. The approach adopted by SLYK is to prioritize dirty blocks by least recently used and flush those blocks first. The idea is that some blocks are going to be more actively written to and read from than others. The ones that are being actively used should be the last to be flushed since there is a good chance that deferring them will save bandwidth. With this priority in mind, SLYK continuously in the background flushes dirty blocks to OpenDHT, but it throttles itself to keep from using too much bandwidth. Only when migration is imminent does SLYK use its full

bandwidth to fully flush out dirty pages.

Figure 2 shows the migration process from computer B to C. After control is transferred to C, B flushes the remainder of its dirty blocks to OpenDHT. Then as C runs, it demands pages in blocks from OpenDHT to cache locally. C suspends on reading a block that B has not flushed yet. Computer A shown in gray has already flushed its dirty blocks.

5.2 File Format

A SLYK disk image has three distinct parts - the header, HD snapshot, and sectors, as shown in the *Original Disk Image* in Figure 3. The header consists of the number of data sectors in the image, and a magic number and version number identifying the disk image as being in SLYK format. The HD snapshot is an array of 20-byte SHA1 hashes of sector contents, each followed by a byte indicating whether or not the sector is stored in the disk image, sorted by sector ID (i.e., each sector has a corresponding SHA1 hash). At the end of the image is a section containing all of the actual sectors, also sorted by sector ID.

When SLYK is bootstrapped with a SLYK-formatted disk image, the HD snapshot section of the Original Disk Image (read-only) is copied over to a *Working Disk Image* as shown in Figure 3. Changes to any sectors are written to the HD snapshot on the Working Disk Image. The Working Disk Image also contains a *sector cache*, an array of sectors that have been cached but not yet flushed to OpenDHT.

5.3 Cache

Caching improves I/O performance by minimizing the amount of disk-bound accesses. For optimization purposes, SLYK keeps a *Sector Lookup Table* in memory. As shown in Figure 3, the table is indexed by sector number and contains the offset into the Working Disk Image's sector cache where a sector's contents are stored.

On reads, SLYK looks for the block in the following order until it finds it:

1. The sector cache on the Working Disk Image
2. The sectors on the Original Disk Image
3. OpenDHT

On writes, the blocks are cached and marked as dirty. Dirty blocks are flushed on a least recently used basis, either when the cache is full or after migration.

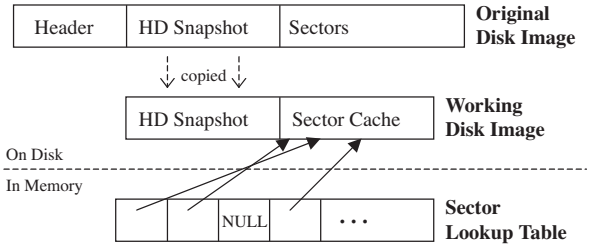

Figure 3: SLYK disk structure. The *Original Disk Image* is a read-only image from which the HD Snapshot is copied over to the *Working Disk Image*. The Working Disk Image acts like a dynamic cache that stores sectors that are indexed by the offsets in the in-memory *Sector Lookup Table*.

6 Network Transparency

Many of the challenges we encountered in designing SLYK branched from our vision for SLYK to be as deployable as possible. There would have been many possible alternate designs to SLYK if we had not been aiming for this property. To achieve this goal of deployability, we wanted to avoid disrupting communication with any nodes on the network that SLYK might wish to talk to, which includes servers and other SLYK nodes. We refer to this property as network transparency. In order for SLYK to be transparent to the network, it needs to emulate a virtual network environment that programs and users will be familiar with, and have mechanisms for migrating active connections without cooperation from the remote machine.

6.1 Virtual Network Environment

SLYK communicates with the outside world as if it were on its own private network behind a NAT. The virtual router, DHCP server, and DNS server are all emulated in software, thus giving SLYK the flexibility to perform high level routing decisions while still presenting a familiar environment to the guest operating system and applications.

The virtual network QEMU simulates includes a virtual router and DHCP server (10.0.2.2), a DNS server (10.0.2.3), and a SMB server (10.0.2.4). The DHCP server allows the guest operating system to autoconfigure itself to the virtual network. The DNS server forwards requests from the guest operating system and applications to the real name resolving mechanism provided by the host operating system. Fi-

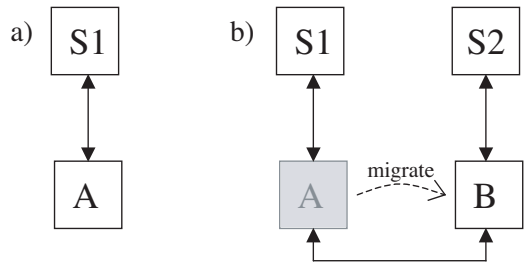

Figure 4: SLYK preserves active network connections through machine migration. (a) Before any migration occurs, machine A connects to sender S1. (b) After migration to machine B, the connection to S1 is indirectly maintained via forwarding through A. B also establishes a direct connection with sender S2.

nally, traffic routed to the virtual gateway are translated into system calls into the underlying sockets or Winsock API of the host. In this way, the entire QEMU system can run non-privileged. This method contrasts with the approach taken by Bochs and VMWare, and does not employ the QEMU option of communicating directly with the kernel using raw packets (which requires superuser status in many operating systems).

6.2 Active Connection Migration

Without cooperation from the remote machine, network connections cannot be fully migrated. There have been several proposals to augment internet routing with a truly mobile system such as i3 [17] or UIP [7]. However, without the widespread deployment of such systems, and given the restriction that no modification can be done to the remote machines for maximal deployability, SLYK is forced to adopt a packet forwarding-like technique such as that used in Mobile IP. However, the solution adopted by SLYK is a little more flexible, since connections to the virtual environment are made mobile without requiring the host's connections to be forwarded. SLYK also treats UDP and TCP traffic differently, being more bold about suddenly moving UDP traffic since it is inherently connectionless.

SLYK is optimized for the common case of the virtual system trying to establish a direct connection with a remote machine. In this situation, a direct connection is established between the host machine running SLYK and the remote machine, and packet rewriting tricks them into thinking they are communicating directly. It is only when there are lingering active connections at the time of migration that forwarding needs to take place.

When there are lingering active connections, the machine that the virtual system migrated from does not fully shutdown SLYK. Instead, SLYK switches into a simple routing mode which keeps the connection alive with the remote machine and forwards any incoming packets to the new location of the virtual system. In this mode, SLYK also forwards packets from the virtual system to the remote machine. Figure 4 shows computer A switching into this routing mode since the connection to S1 was lingering at the time of migration.

This system does not maintain a perfect illusion of active network migration. For example, a forwarding machine may go down. This would appear to the virtual system the same as if the remote machine dropped the connection. In many situations, the interested application or operating system may attempt to reestablish the connection and would succeed in directly connecting with the remote machine.

7 Evaluation

We evaluated SLYK by verifying correct operation and taking performance measurements. Several combinations of different host and guest operating systems were installed and successfully tested running SLYK. To evaluate the performance of our system, we ran benchmarks for CPU-intensive and disk-intensive tasks and took timing measurements for network-oriented tasks such as migration.

7.1 Verification

We installed SLYK on Linux, Windows XP, and Mac OS X host platforms, and on each of the hosts, we tested running SLYK with Windows 2000, Debian, and Knoppix as guest operating systems. On Debian and Windows, we tested certain applications that were pre-installed in the disk images to see if their behavior was normal. In both cases, ssh, AIM, and Firefox loaded successfully and were able to connect to the Internet. In addition, we were able to create and run a PowerPoint presentation in Windows, and create and save Emacs files in Debian.

Migration tests were performed with both the Windows and Debian guest systems running on the various host platforms. In each case, one or more applications were loaded on one machine and successfully

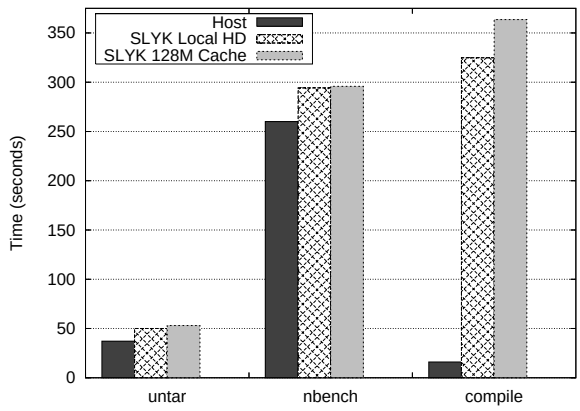

Figure 5: Timing results for nbench, untar, and compile on a Linux host machine compared to Debian running on SLYK with various configurations.

migrated to the second machine. For example, using Firefox on Debian, we installed Flash, loaded a Flash animation in the browser, started playing it on the first computer, migrated to the second computer, and resumed the animation from where it left off. We also successfully migrated files that were saved to the hard disk of the first computer.

7.2 Performance

We ran three benchmark tests to quantify the performance of the SLYK virtual system compared to the host machine. The tests were run on a Pentium 4 2GHz desktop computer running RedHat Enterprise Linux WS, with GCC version 3.2.3 and the 2.4.21 Linux kernel. The guest O/S on which we tested SLYK was Debian Testing (3.1), running on the same host machine. For the first benchmark we downloaded a tar file of version 2.6.11 of the Linux kernel [20] and measured the time it took to untar the file (a disk-intensive workload). The second benchmark was the Unix port of the nbench benchmark suite [4, 11], which tests CPU-intensive workloads. For the third benchmark, we downloaded version 0.7 of XmlRpc++ [12] and measured the time needed to compile the XmlRpc++ RPC library, a mixture of CPU and disk-intensive workloads. For all of the measurements, we used the Unix *time* function to measure time on the host machine and a stopwatch on the guest machine (since the output of *time* on the VM does not correspond exactly to real time). Average times of five runs of each benchmark are shown in Figure 5.

	Avg. Time (sec)
Migrate	27.36
Slow Migrate	59
Full Migrate	52.5
DHT Put	0.53
DHT Get	0.42

Table 1: Timing results for migration of Debian between a Linux host and Windows host, and the average time for DHT put and get accesses.

For the first benchmark, we expect the difference between emulated performance and native performance to be similar, since the disk is the bottleneck for both systems. The second benchmark was a little more surprising since we expected emulated performance to be poor for CPU-intensive tasks. The closeness of emulated to native performance is a testament to the QEMU dynamic code translation engine. Upon examination of the benchmarks, the reason QEMU did so well became more clear. The benchmarks were typically algorithmic, loop-heavy benchmarks which would utilize QEMU's translated code cache very well. The last benchmark is the most stark. It is a complicated build benchmark that exercises both communication and computation. Unfortunately, QEMU currently does not overlap communication and computation as well as it could. QEMU is a single-threaded program with one main loop. In several places it resembles an event-driven design, but in other places the entire thread blocks when it should not have to. The closeness of SLYK running on the local HD and SLYK running on the DHT store with 128M cache shows that our DHT store implementation has been optimized enough to be competitive with normal QEMU performance though.

Next, we measured the time it took to perform various types of migration between the same Linux machine and and a Pentium M 1.6GHz laptop running Windows XP, connected by a 100Mbps switch. A "slow" migrate used our initial implementation of migration, a regular migrate was a much more efficient version that used sockets, and a "full" migrate consisted of the regular migrate along with all of the state required to reconstruct the user's hard drive. Table 1 shows a summary of the migration times. In addition, for reference we included the average time it took to perform a single DHT *put* or *get*.

8 Future Work

Significant improvements can still be made to SLYK's performance, fault tolerance, and deployability. SLYK is based on a few projects and inherits some of their good qualities and shortcomings. There are also similar projects to SLYK with optimizations and ideas worth borrowing.

One obvious technique to speed migration is to compress the memory image before transferring from source machine to destination machine. However, compression may require excessive CPU time in an environment where the CPU may already be taxed. Compression might not turn out to be an overall win unless the memory image is readily compressible. One clever technique, termed ballooning [15], involves writing a device driver that integrates unobtrusively with the guest operating system which, immediately before migration, requests many pages from the operating system, causing it to page out all not-recently used pages. The driver then clears the pages it requests to all zeros, causing the resulting memory to be highly compressible. This technique unfortunately does require something written specifically for the guest operating system. However, the specifics of requesting large regions of memory and clearing them to zero are usually alike in many operating systems.

Many improvements can be done to speed up SLYK's steady state performance as well. QEMU already does a very good job with its dynamic code translation engine for heavy CPU code with many repeating regions. It also does well on purely I/O bound tasks since native execution would share the same bottlenecks. However, in our experience QEMU cannot effectively overlap computation with communication, resulting in greatly degraded performance on benchmarks which exercise both. This was especially bad for us when the latency of contacting a DHT was adding into the mix. To have any acceptable performance, we needed to do aggressive caching and some tweaking of QEMU's main event loop to make more operations non-blocking. Our performance tweaks have allowed SLYK running on a DHT-backed store to approach the performance of QEMU running entirely out of disk, but much more can be done to approach native execution speeds.

SLYK can also better utilize being connected to a highly reliable storage to perform automatic backing up of state. The hard part in integrating a DHT-backed store throughout the SLYK code has already been done. Furthermore, SLYK already uses hash arrays as compact snapshots of the current HD stage. The next step would be to utilize this link to perform automatic and consistent backups of the SLYK system. Many ideas can be borrowed from archival systems such as Venti-DHash [16] or the many log-based systems in existence. SLYK was originally designed with backup in mind, so adding it in should not be difficult. However, tackling the original degraded performance when running with a DHT became a more pressing concern.

With a great deal of effort, we have managed to produce a project that is both portable and deployable. This involved needing to deal with the different quirks of every platform and trying to cut down on the number of external dependencies our project relied on. In the end we feel we have been fairly successful in this effort, but now with our experience and a little bit of redesign, SLYK could potentially integrate even better with user-preferred computing environments.

9 Conclusion

SLYK attempts to offer strong notions of fault tolerance and mobility, including machine transparency, network transparency, and active connection migration. It does so at a performance cost in order to ensure compatibility with as many operating systems, applications, and computing environments as possible. Our prototype demonstrates that despite this overhead, a working, usable system can be built that offers the illusion of full mobility.

Many optimizations can be performed to bring down the cost of mobility. Some, such as dynamic binary translation, gzip compression, and ballooning, have already been explored. Others may benefit greatly from hardware, operating system, and network support. The costs associated with mobility will continue to decrease until one day full user mobility may become a normal part of our computing environment.

Acknowledgments

We would like to thank Robert Morris and Athicha Muthitacharoen for their suggestions and guidance. We would also like to thank the developers of QEMU and XmlRpc++ for producing their quality projects. Finally, we would like to give thanks for the OpenDHT folks and the maintainers of the many

PlanetLab nodes we hammered for putting up with our traffic and not dropping our data.

References

[1] M. Beck, T. Moore, and J. S. Plank. An End-to-End Approach to Globally Scalable Network Storage. Technical Report PDN–02–007, PlanetLab Consortium, November 2002.

[2] F. Bellard. Qemu CPU emulator User Documentation, 2003.

[3] F. M. T. Brazier, B. J. Overeinder, M. van Steen, and N. J. E. Wijngaards. Agent factory: generative migration of mobile agents in heterogeneous environments. In *SAC*, pages 101–106, 2002.

[4] BYTEmark. http://www.byte.com/bmark/bmark.htm.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[6] D. Eastlake. Us secure hash algorithm 1 (sha1). The Internet Society: RFC 3174, September 2001.

[7] B. Ford. Unmanaged Internet Protocol.

[8] IBM Corporation. *IBM virtual machine facility/370: planning guide Publication Number GC20-1801-0*, 1972.

[9] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 40, Washington, DC, USA, 2002. IEEE Computer Society.

[10] K. Lawton, B. Denney, N. D. Guarneri, V. Ruppert, C. Bothamy, and M. Calabrese. Bochs x86 PC emulator Users Manual, 2003.

[11] Linux/Unix nbench. http://www.tux.org/~mayer/linux/bmark.html.

[12] C. Morley. XmlRpc++, 2002.

[13] J. Nieh and O. C. Leonard. Examining VMware. *j-DDJ*, 25(8):70, 72–74, 76, Aug. 2000.

[14] C. E. Perkins. IP Mobility Support for IPv4. Internet Engineering Task Force: RFC 3344, August 2002.

[15] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[16] E. Sit, J. Cates, and R. Cox. A DHT-based Backup System, 2003.

[17] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure, 2002.

[18] N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith. NOMADS: Toward a Strong and Safe Mobile Agent System. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 163–164, Barcelona, Catalonia, Spain, 2000. ACM Press.

[19] B. K. Sylvia. Spurring adoption of dhts with openhash, a public dht service.

[20] The Linux Kernel Archives. http://www.kernel.org/.

[21] T. Walsh, P. Nixon, and S. Dobson. As strong as possible mobility: An Architecture for stateful object migration on the Internet.