Migration without Virtualization

Michael A. Kozuch, Michael Kaminsky, Michael P. Ryan
Intel Research Pittsburgh

Abstract

Migrating a live, running operating system from one machine to another has proven to be an invaluable tool over the past few years. Today, however, the only way to migrate an OS is to run it in virtual machine, thereby incurring the disadvantages of virtualization (e.g., virtualized devices often do not keep pace with the latest hardware). This paper proposes a new infrastructure for operating systems to allow direct migration from one physical machine to another physical machine—even if the hardware on the target machine differs from the source. We believe that this approach can be viable and practical as many modern operating systems already provide the initial support necessary through their hibernate and suspend power management infrastructures.

1 Introduction

In 2001, Chen and Noble made a case for virtualization by proposing three example services [3]. One of their examples—environment migration—has since become commonplace and an essential feature in many environments.

Although Chen and Noble refer to environment migration in the context of moving a user's environment from one client machine to another, similar techniques have been used to move computation within a data center [4, 7]. Migration within a server room provides a number of advantages. For example, administrators might want to migrate heavily communicating machines near each other to reduce network load, migrate an environment from a failing machine, migrate computation from an under-powered machine to a more powerful one as load increases, or migrate environments to consolidate computation onto fewer data center racks to enable some racks to be powered down completely.

In recent years, many of us have accepted the argument that virtualization is the best solution for environment migration, *prima facie*. Of course, prior to the interest in virtualization, several proposals were put forward to enable *process migration* [5, 2, 8]. This paper asks: why not build the migration infrastructure directly into the operating system?

1.1 Why not process migration?

Moving computation from one platform to another by using *process migration* has been well-studied [5, 2]. However, effecting a practical process migration system has proven challenging because, while the OS isolates user processes from the platform, applications tend to have many connections to the OS that are tricky to migrate (e.g. sockets, file descriptors, etc.). Additionally, processes may interact with other processes through shared memory, etc. Although recent work has proposed a solution that includes the notion of *process domains* [8], migration between two machines still requires (1) *a priori* partitioning of processes into domains, (2) extensive system support, and (3) that the OS images and libraries be virtually identical on the two machines.

Rather than migrate a domain of running processes from off of an operating system, we propose to migrate the OS (with the processes) from the hardware.

1.2 Why not virtualization?

The typical solution for migrating a running OS from one platform to another is to leverage virtualization technology [10, 4]. However, this technique requires an extra layer of abstraction that introduces several disadvantages:

- **Capability Lag.** VMMs typically expose "lowest common denominator" virtual devices to enhance portability. These devices typically fail to expose the highest performance features of the physical devices present. Further, to take full advantage of physical devices, up to three modules must work in concert: the device driver in the VMM, the virtual device model in the VMM, and the device driver in the guest OS. The difficulty inherent in ensuring this stack works together to provide efficient access to the device implies a high probability that a system performs less than optimally.
- **Software Management.** Although virtualization proponents claim that VMMs are sufficiently small to introduce little additional complexity into software management, virtualization typically does not reduce the total number of software components running on a system. Hence, there are more lines of code to manage, more patches to apply, etc.

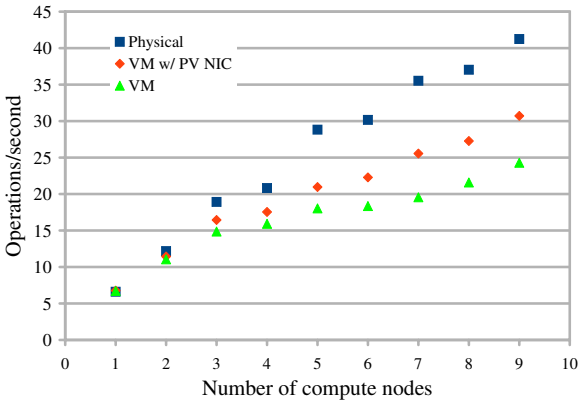

Figure 1: Performance of DPRSim2, an MPI-based parallel application, when running (i) on physical hardware, (ii) in a virtual machine with a para-virtualized NIC, and (iii) in a virtual machine with fully virtualized NIC. The compute nodes are conventional rack-mounted, 8-core, Linux 2.6 servers with 1 Gbit/s Ethernet.

- **Performance.** In many applications, virtualized performance is within an acceptable margin of native performance, and therefore, the additional layers of software introduced through virtualization are tolerated, but there are also cases where it is not [9]. For example, Figure 1 shows the performance overhead that virtualization imposes on a parallel robotics simulator [1]. The application is based on MPI and makes frequent calls to barrier primitives. Running in a standard VM configuration, it suffers a 40% slowdown; even with a para-virtualized NIC, the slowdown is still 25%.

1.3 Our Proposal: OS Migration

This paper proposes a set of operating system modifications to allow live migration of a running OS to new hardware without virtualization. That is, our basic goal is to take an OS running on the bare hardware of one machine and move it to another physical machine. The hardware on the target machine may differ from that of the source machine; furthermore, the migration must be "live" in the sense that user-space processes running prior to migration must continue running after migration. That is, rebooting is unacceptable. However, as shown in Figure 2, the OS migration process will re-initialize the state of the devices (and other relevant parts of the OS) to handle any difference between the source and target platforms. Furthermore, many modern operating systems already provide initial support for this process through their hibernate and suspend power management infrastructures.

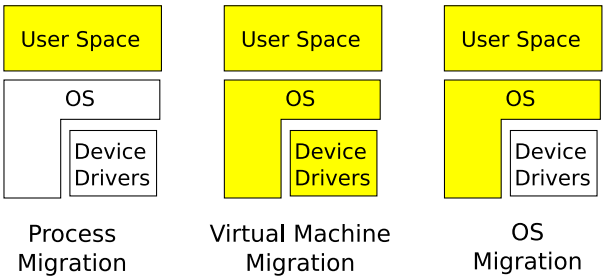

Figure 2: Shading indicates what migrates in process, VM, and OS migration.

2 Design Space for Migration

We can divide operating system migration into three categories based on the target machine (where the OS is migrating to). Similarly, we can categorize migration based on the amount of state encapsulated and the time scale associated with migration.

Target Platform Migration involves moving the operating system from a *source* machine to a *target* machine. The greater the difference between the source and target machines' hardware platforms, the greater the complexity in moving the OS. We identify three basic scenarios:

- **Same machine.** One could imagine a migration where the source and target machines not only have the same hardware but are literally the same machine. This migration scenario essentially describes what happens during hibernation in today's operating systems.
- **Different machine; same hardware.** The next category is where the target machine is physically a different machine, but it has the same hardware as the source. Here, the complications come from non-deterministic aspects of the boot process (e.g., the BIOS and/or the OS might enumerate the devices in a different order or use different IRQs).
- **Different machine; different hardware.** The most complex migration, not surprisingly, is when the source and target hardware differ. Here, the migration process must somehow encapsulate the state on the source in a device-independent way, move it to the target machine, and map that state to the new set of devices (where possible). We discuss this scenario at length in Section 3.

Time scale and encapsulation A second aspect of the migration design space is the time scale over which the migration takes place and how much state (if any) it encapsulates. We identify four points in this axis of the design space:

- **Shutdown/reboot.** The most basic form of "migration" is simply to shutdown the source OS and reboot it on the target machine. If the operating system kernel has drivers for the target machine's hardware, the only requirement is to move the disk to the target machine. In many cases, the disk might be available on the network (e.g., the machine is in a data center and boots from a network drive). This type of migration works trivially today but discards all of the active state that we would like to keep.
- **Hibernation.** The next point in the spectrum is to migrate the running operating system environment to the target machine without discarding state. Hibernation, available in most modern operating system, involves saving the entire system state to disk and re-loading that state on resume. Today, however, hibernation is designed to work on the same machine, though extending hibernate/resume to a different target machine with the same hardware is probably a relatively simple extension. Hibernation, however, does not work across different hardware profiles; furthermore, hibernation is not live—the assumption is that certain state (e.g., network connections) will not exist after resume.
- **Suspend.** Suspend to RAM involves putting the devices and operating system into a low power state. Today, suspend only works on the same machine, and like hibernation, suspend assumes that the hardware remains the same. Both suspend and hibernate explicitly freeze all processes before activating their respective modes.
- **Live.** Live migration implies moving the operating system from the source to the target without explicitly telling the OS to freeze processes. Live migration is used in several virtualization environments to move guest VMs from one VMM to another—usually with a downtime of less than a second. In virtualization, however, migration only works when the source and target VMMs present the same hardware profile. We hope to enable live migration from a source to a target where the hardware profiles are different.

In summary, several of the points in this design space are already possible today: "migration" from and to the same machine is already possible using standard OS power managements facilities. Migrating an OS to a different machine with the same hardware might be possible today through hibernation (rebooting is also possible, but uninteresting). Live migration from a source to a target even with the same hardware is not possible today, let alone different hardware—which is our goal.

We note that migration through virtualization actually collapses the first axis of the design space. The two "different machine" scenarios do not occur because the VMM presents exactly the same hardware profile to the OS regardless of the underlying physical machine. With virtualization, live migration is possible today, but suffers from the limitations discussed in Section 1.2.

3 Migration-Enabled OS

In this section, we explore what changes must be introduced into a modern OS such as Linux to enable migration. We first visit the challenges that must be overcome and then our proposed solutions.

3.1 Challenges

At the beginning of the migration process, the source machine is running the operating system and programs that must be moved. The target machine is off or in a state that can be controllably rebooted. In such an environment, five problems must be solved in order to perform a successful migration.

Lack of Receiver The target machine must have an operating environment that allows the source machine to transmit machine state to it and to transition into the operating system once received.

Dynamic shutdown/startup The source machine needs to be able to cleanly shutdown any devices that may be attached to it. In addition, the target machine must be able to initialize or reinitialize any devices into a state that corresponds to the state of a matching device on the source machine. Note that this is not hotplugging; the kernel serves notice that a migration event is pending, so device drivers are given an opportunity to cleanly shutdown and initialize devices.

Device state abstraction If the source and target machines have hardware devices that differ, even slightly, the two systems will need to negotiate a way to map the source machine's hardware resources into those of the target machine. Current systems that suspend, resume, and migrate do not offer direct solutions to this problem. VMMs hide differences in hardware platforms, and software suspend and hibernation on present operating systems assume that the same machine will be there at resume. In Linux, for example, it is not possible to simply swap a network card between software suspend and resume without manually disabling the PCI device before suspension and forcing a bus scan after resume. Both of these techniques rely on the identical presentation of the underlying hardware.

Self-migration In order to perform a live migration from the operating system, it is necessary to obtain a consistent view of the entire system from within it. If, for example, the *pre-copying* algorithm [11] is used, then the pages

that are used by the system to prepare and transmit dirty pages to the target are impossible to freeze and transmit.

Post-migration patchup Once the state of the source system has been transferred to the target, it is necessary to fix or patchup some of the system state in order to have it match the source system. For example, the target machine's Ethernet card's MAC address might need to be updated to match that of the source in order to migrate active network connections from the source machine.

3.2 Solutions

Lack of Receiver One straightforward way for the target machine to receive the source system's state is to use a bootstrap kernel. This kernel could come from a central repository of kernels or some other form of boot-loader, such as PXE booting. Once it receives the source machine's kernel, the bootstrap kernel can switch into it using *kexec*. The target machine could even download the source machine state through a user-space process using a mechanism that is similar to user-space software suspend in Linux.

Dynamic shutdown/startup Devices on the source and target machine can be shutdown and initialized by leveraging the currently existing suspend and resume calls in the Linux kernel. It may, however, be necessary to update the routines implemented by a device driver to support resuming from a partially initialized state (e.g., when a network card is used to receive the source machine's state).

Device state abstraction Migrating from one device to another requires a mapping between them. This mapping can be achieved by providing a uniform device descriptor for each class of device. For example, network card drivers could implement `export()` and `import()` routines that export and import descriptions of the device that other drivers of the same class could interpret (e.g., the MAC address and the MTU size set for the card). The procedure might work as follows:

- **Export.** During the process of migrating, the source machine calls `pdev->export()`, which returns a data object that contains information about the device exported by the device driver, saving the output from this call for transmission to the target machine.
- **Suspend.** The source machine completes transmitting its state to the target and calls `pdev->suspend()`.
- **Import.** The target machine calls `pdev->import()` with the data object that was returned by `export()` on the source machine, populating its internal data structures and fields with the information.
- **Resume.** The target calls `pdev->resume()`, initializing the device to a state that matches the data in the device driver's internal structures.

As mentioned above, it may be necessary to update the resume routine to support resuming from a partially initialized state. It is also possible to extend device class structures (i.e., `net_device`) to contain this information, but it would be necessary for device drivers to be updated to properly interface with this new structure in order for such a scheme to work.

Self-migration To facilitate migration, we propose that the OS categorize memory pages according to four types: (a) kernel code and read-only data structures, (b) device independent kernel data structures (including the process tables and page tables), (c) device-dependent data structures, and (d) user-space memory. Category (a) may be re-established on the target machine either by shipping it to a bootstrap kernel or by delivering a copy from another machine using PXE, as proposed above. Both (b) and (d) can use the previously mentioned *pre-copying* algorithm—either leveraging the newly-installed kernel at the target or a bootstrap kernel. Category (c) involves transferring the exported device state and then importing it during the *post-migration patchup*. Similar to [6], a consistent final view of the system may be achieved by performing resend-on-write followed by a copy-on-write.

Post-migration patchup The patchup performed at the target host is largely accomplished by the pair of calls to import and resume, which populate device driver structures and reinitialize devices to a known state. It will also be necessary to merge device dependent state and device independent state in some core kernel data structures as a final step before resuming execution.

3.3 Assumptions

Our design for server migration is based upon a number of assumptions. First, because OS images may resume on any machine in the cluster, we assume each image contains the necessary device drivers for each physical platform. Second, the OS must be modified to enable the identification of the different page types identified under *Self-migration*, above. For example, Linux might expose a new version of *kmalloc()* that accepts a typing parameter. Finally, we've assumed that no devices are visible in user-space (i.e. the OS abstracts the platform). In practice this might not be true. For example, we've assumed that the source and target processors are from the same vendor, but there may still be ISA differences (e.g. vector SIMD instructions). Another example is user-space-visible device names (e.g., network and block devices). In such cases, there are several coping strategies. Some features could be encapsulated in libraries or device drivers that

expose an abstracted interface allowing migration, the OS can refuse to migrate to a less-featured platform, the OS could trap on faults and migrate to a more featured machine, or OSes could choose a limited feature-set on which to standardize.

3.4 Special devices

Resizing memory In order to support resizing memory, we propose using some of the mechanisms that are already in place to minimize the size of a software suspend image. This includes flushing the buffer cache and swapping out (if possible) pages to disk.

Number of processors If the number of processors is different on the target machine than on the source machine, the kernel can migrate processes as necessary. Modern operating systems, such as Linux, already support this capability through CPU hotplugging.

4 Discussion

Additional benefits We believe that this architecture can also provide several additional benefits. For example, besides migrating an OS from one physical machine to another, one could use OS migration to move from physical to virtual or vice versa. The VMM simply presents a hardware platform like a physical machine would. Migrating from physical to virtual is particularly useful in data center settings when an administrator wants to consolidate resources. He or she could use OS migration to move a set of OS images running on independent physical machines onto a single (presumably, more powerful) host to save power when the machines are largely idle.

Likewise, an administrator of a compute cluster might encourage users to run virtual most of the time to maximize flexibility, but occasionally migrate to a physical node in order to run performance benchmarks or, in the case of a service, to handle a flash crowd.

Disadvantages As noted in Section 1, Chen and Noble identify two benefits of virtualization in addition to environment migration: secure logging and intrusion prevent/detection. These benefits, however, stem from the virtualization infrastructure itself and are not related to VM-based migration.

Migration through virtualization does, however, provide consolidation as an additional side benefit. Our OS migration does not directly support consolidation since it inherently pairs one OS per machine. As noted above, however, OS migration can support consolidation by allowing the OS to migrate from a physical to virtual machine, which can then be consolidated.

Another potential disadvantage of our OS migration proposal is complexity. Virtualization has the benefit that it automatically provides migration for unmodified guest operating systems whereas OS migration requires a modest, but new OS migration infrastructure. Once the initial hooks are available, though, one could imagine—like suspend/resume—that migration would become a commonly-supported OS feature.

5 Conclusion

The ability to migrate a running operating system to another machine with different hardware has several benefits. This paper proposes an operating system infrastructure to enable such a migration without virtualization. The power management infrastructure found in modern OSes already provides a number of hooks that help make migration feasible with manageable OS changes.

Acknowledgments

We thank Michael Ashley-Rollman and Babu Pillai for help with DPRSim, as well as Dave Andersen and our reviewers for their helpful comments.

References

[1] DPRSim: The Dynamic Physical Rendering Simulator. http://www.pittsburgh.intel-research.net/dprweb/.

[2] A. Barak. The mosix multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13:361–372, 1998.

[3] P. M. Chen and B. D. Noble. When virtual is better than real. In *HotOS*, 2001.

[4] C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI*, 2005.

[5] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software—Practice and Experience*, 21:757–785, 1991.

[6] J. G. Hansen and E. Jul. Self-migration of operating systems. In *ACM SIGOPS European workshop*, 2004.

[7] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX ATC*, 2005.

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

[9] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59R1, HP Labs, 2007.

[10] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, 2002.

[11] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the v-system. In *SOSP*, 1985.