

Self-migration of Operating Systems

Jacob Gorm Hansen & Eric Jul
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
{jacobg,eric}@diku.dk

1 Introduction

This paper is about on-the-fly migration of entire operating systems between physically different host computers. Resource allocation is often static; using migration allows applications to dynamically change bindings between programs and physical hosts as to improve utilisation. We first survey different approaches to migration and then present two prototypes that allow migration not only of an application but also of the operating system running the application. One of the prototypes includes a novel approach, *self-migration*, to operating system migration. Performance numbers show that migration can be done with merely subsecond suspension of the application.

One of the motivations for our work is the growing popularity of large-scale clusters of commodity computers for scientific computation. In such systems, job submission and scheduling are most often centralised and static, similar to how mainframes were used before the advent of interactive multiprogramming systems. For example, in a large cluster that we know of, it is not uncommon for a customer to have to wait weeks before a parallel computation involving a majority of the hosts can be run, because freeing up all hosts means waiting for every job in the system to complete. Such static scheduling is an impediment to efficient utilisation of clusters.

Two other major impediments to the use of large grids of clusters are the lack of agreement on a standard software configuration, and, for security reasons, the restrictions on private users concerning job submission and user contributed resources.

Process migration has previously been suggested as a way of addressing scheduling problems in distributed systems, but due to a number of inherent problems, has not caught on. In this work, we investigate the feasibility of changing the unit of migration from a single

process to the entire operating system. Our hypothesis is that operating system migration is a viable solution to the problem of dynamic scheduling in large-scale computational networks, and that it addresses significant configuration management and security issues as well.

To investigate this approach we have implemented two prototype systems, one based on the L4 microkernel, and one based on the Xen hypervisor. Due to the differences in the underlying architectures, the prototypes implement migration in radically different ways, but common to both is that support for operating system migration adds no overhead to execution, and that the downtime experienced when migrating is negligible.

In the following, we present three approaches to migration and then present our two different prototypes. We briefly discuss performance and configuration issues before presenting our major conclusion that our migration approaches both are viable and effective.

2 Approaches to migration

In the 1980s several process migration systems were developed. Such systems usually suffer from dependencies on the source host OS, so-called *residual dependencies*. Systems such as Sprite [1] and MOSIX [2] suffer from such residual dependency [3] problems, preventing their use across wide-area networks. Zap [4], solves this problem by grouping processes in closed *capsules*, preventing reliance upon extra-capsular entities, so that capsules can migrate freely. All three systems mandate a uniform operating system configuration across all participating hosts, something which is hard to enforce across institutional barriers. Figure 1 depicts a typical process migration scenario with residual dependencies that arise when the migrated process must cross traditional security and configura-

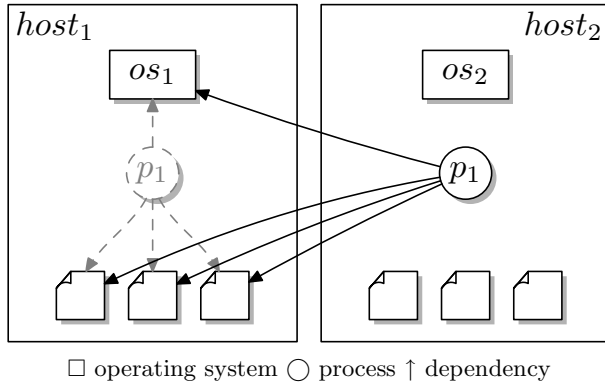


Figure 1: Process migration example. p_1 has migrated from $host_1$ to $host_2$.

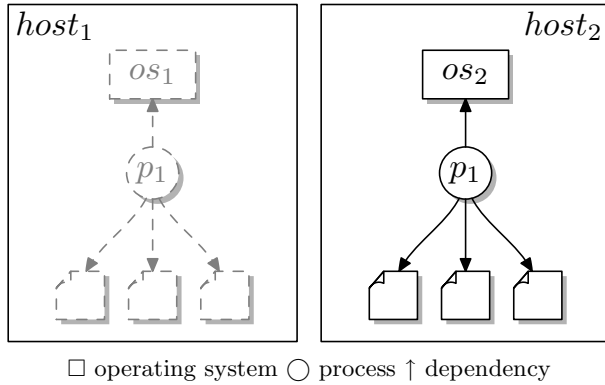


Figure 2: Migrating p_1 from $host_1$ to $host_2$ by means of operating system migration.

tion management boundaries between distinct operating systems.

A second way is to use object mobility or mobile agent frameworks, e.g., Emerald [5] or Telescript [6], but apart from the problem of having to rewrite a huge body of existing code to run within such frameworks, such systems must be fairly feature-complete to be generally useful. As a minimum, they should support concurrent execution, code reuse, and protection and fault isolation between components and users. It is likely that the feature set of a given mobility framework will converge towards that of a traditional operating system.

Seen in this light, turning operating systems into mobility frameworks may be more practical than turning mobility frameworks into operating systems, which leads to a third approach to mobility.

Our third way of migrating an application is to move *the entire operating system* including all contained processes, file systems, and network connections. Apart from its role as a hardware abstraction layer, the OS may be viewed as a shared library servicing the sharing

and protection needs of user applications, and, as with traditional shared libraries, should be made available in exactly the same version on the destination host of a migrated application. Moving the OS along with the application solves a range of configuration and security problems, because the OS encapsulates such issues, and because the traditional boundaries protecting an OS against intrusion still apply. Figure 2 shows a migrating operating system. In contrast to the process migration example of Figure 1, OS migration maintains the traditional security and management boundaries, and thus, apart from the need for protecting the OS while in transit, introduces no new liabilities in these areas. Next, we describe our two prototypes that implement OS mobility.

3 Host-driven migration

Our first implementation, NomadBIOS [7], is a prototype host-environment for running several adapted Linux-instances concurrently, and with the ability to migrate such instances between hosts without disrupting service, what we call *live-migration*.

NomadBIOS runs on top of the L4 microkernel [8], handing out resources to zero or more concurrently running *guest operating system* instances. It contains a TCP/IP stack for accepting incoming migrations and for migrating guest operating systems elsewhere. NomadBIOS starts an incoming guest operating system as a new L4 task in its own nested address space, and provides it with backing memory and filtered Ethernet access. The guest OS is an adapted version of L4Linux [9] 2.2, called NomadLinux, binarily compatible with native Linux. All guest memory is paged by NomadBIOS, so it is simple to snapshot the memory state of a guest OS and migrate it to another host, though quite a lot of cooperation from the guest is needed for extracting all thread control block state from the microkernel.

To reduce downtime, NomadBIOS uses pre-copy migration [10], keeping the guest OS running at the originating host while migrating, tracking changes to its address space and sending updates containing the changes to the original image over a number of iterations. The size of the updates typically shrinks down to a hundred kilobytes or less. As a further optimisation, a gratuitous ARP packet is then broadcast to the local Ethernet, to inform local peers about the move to a new interface. For migrations crossing subnets, the guest OS must change its IP address, or it can use an IP-indirection layer such as Mobile IP [11]. Currently, files are accessed via iSCSI or NFS, and so remain available after migration. Future work will

address migration of block device contents along with the OS. Because the bulk of the migration work is handled by the host-environment, we refer to this setup as implementing *host-driven* migration.

NomadLinux running under NomadBIOS was benchmarked [7] against Linux 2.2 running under VMWare Workstation 3.2,¹ and against a native Linux 2.2 kernel where relevant. Performance was generally on par with VMWare, and scalability when hosting multiple guests markedly better. When compared to native Linux, the performance was equal to that of L4Linux, a 5-10% slowdown for practical applications. Migration of an OS with multiple active processes, over a 100 megabit/s network, typically incurs a downtime less than one tenth of a second.

4 Self-migration

Our second prototype is based on the Xen [12] virtual machine monitor (or *hypervisor*) which divides a commodity Intel PC into several virtual domains, while maintaining near-native performance. Xen derives much of its impressive performance from the lack of comfortable abstractions, such as the paging-via-IPC mechanism and recursive address spaces of L4, and provides no clean way of letting a guest OS delegate paging responsibilities to an external domain. However, this is required when implementing host-driven live-migration as otherwise there is no way of tracking changed pages from the outside. While the Xen team is working to add such functionality, another approach may prove interesting: In line with the end-to-end argument [13], we propose performing the guest OS migration entirely without hypervisor involvement, by letting the guest OS migrate itself. Compared to host-driven migration, this *self-migration* approach has a number of advantages:

Security By placing migration functionality within the unprivileged guest OS, the footprint and the chance of programming errors of the trusted computing base is reduced. For instance, the guest will use its own TCP stack, and the trusted base needs only provide a filtered network abstraction.

Accounting and performance The network and CPU cost of performing the migration is attributed to the guest OS, rather than to the host environment, hereby simplifying accounting. This has the added benefit of motivating the guest to aid migration, by not scheduling uncooperative (for instance heavily pagefaulting) processes, or by flushing buffer caches prior to migration.

Flexibility By implementing migration inside the guest OS, the choices of network protocols and security features are ultimately left to whoever configures the guest OS instance, removing the need for a network-wide standard, although a common bootstrapping protocol will have to be agreed upon.

Portability Because migration happens without hypervisor involvement, this approach is less dependent on the semantics of the hypervisor, and can be ported across different hypervisors and micro-kernels, and perhaps even to the bare hardware.

The main drawback of self-migration is that it needs to be re-implemented for each type of guest OS. While implementing self-migration in Linux 2.4 proved relatively simple, this may not generalise to other systems.

Our self-migrating OS is based on the existing Xen-port of Linux 2.4, known as XenoLinux, which we have extended with functionality allowing it to transfer a copy of its entire state to another physical host, while retaining normal operations. The technique used is a synthesis of the pre-copy migration algorithm and of the technique used for checkpointing in persistent systems.

A traditional orthogonal checkpointing algorithm creates a snapshot of a running program corresponding to the entire state of the program at the instant, the checkpointer is invoked. This is relatively simple, if the checkpointee is subject to the control of an external checkpointer, as the checkpointee can be paused during the checkpoint. If the checkpointee is to be allowed to execute in parallel with the checkpointer, a consistent checkpoint can be obtained by revoking all writable mappings and tracking page-faults, backing up original pages in a *copy-on-write* manner, as described in [14].

Basically, migration can be performed simply by revoking all writable mappings, copying all state, and resending any dirty pages, in a *resend-on-write* manner, until the working set has been identified, and the remaining dirty pages can be transferred after the checkpointee has been suspended on the originating host.

However, in a self-migrating system, transferring the final state is impossible, because the transfer itself would have to alter this state, which would then no longer be the final state. Instead, we can combine the two approaches, performing resend-on-write until the working set has been identified, followed by copy-on-write. After the resend-on-write phase, a partial checkpoint will have been transferred to the destination host, and the remaining changes exist in the pages backed up at the originating host. Once these pages

¹At the time the most recent version.

have been transmitted, the state at the destination equals that at the source at the end of the resend-on-write phase and before copy-on-write. At this point, we may resume execution from the checkpoint transmitted to the destination host, purge the checkpointee from the originating host, and migration is complete.²

Note that if the checkpointee is not purged from the originating host, the result will be a network `fork()`, rather than a migration.

We currently have a working implementation of a self-migrating XenLinux 2.4.26 system, using the technique described above. The working set when migrating an otherwise idle system, over a 100 megabit/s network, is roughly 100 dirty 4 kilobyte pages. The resulting downtime is less than 40 milliseconds.

Initial versions of the self-migration implementation added an extra kernel thread to the guest Linux kernel. This thread would open a TCP connection to the destination host, copy all memory pages, followed by a number of working set patches, and concluded by the set of pages backed up by the final copy-on-write phase. After resumption on the destination host, this thread would block forever, while the originating guest OS instance would simply halt its virtual machine. Due to this behaviour, normal threads and processes were allowed to continue execution on the new host. However, this approach was inflexible, as most of the migration parameters (TCP connection, unencrypted, halt source OS after completion) were hard-coded inside the guest OS. Instead, a new device node `/dev/checkpoint`, from which a user process can read a consistent checkpoint of the machine, was implemented in XenLinux. The data read from this checkpoint corresponds to the data transmitted by the special kernel thread when migrating, a list of (page frame number, page data) pairs, with each page in the system appearing one or more times. The user process performs the actual socket setup, and writes the checkpoint data to this socket. The device `read()` operation signifies end-of-file in different ways, depending on which side of the network fork it is running, so that the appropriate action can be taken. On the source host, `sys_reboot()` is called to destroy the guest instance, and on the destination host a simple arrival message is printed to the user. With this approach, most migration choices are deferred into user space, enabling a user to tailor the migration protocol, and to implement network forkings simply by not destroying the source instance, and reconfiguring the IP address of the destination instance upon arrival. Figure 3 lists the main loop of the user mode migration driver.

²For completeness, all network traffic unrelated to migration should be firewalled off after the resend-on-write phase, as otherwise network peers will be confused.

```

int sock = socket (...);
int f = open("/dev/checkpoint",
              ORDONLY);

do {
    char page[sizeof(int)+4096];
    bytes = read(f, page, sizeof(page));
    if(bytes < 0) {
        printf("we have arrived\n");
        exit(0);
    }
    write(sock, page, bytes);
} while(bytes > 0);
sys_reboot();

```

Figure 3: User space migration driver. Because the kernel forks below the user space process, the program will terminate with both the `exit()` and `sys_reboot()` statements, depending on the side of the connection. Implementing network forking or checkpointing through this interface is straightforward.

5 Implications

The fact that we can now perform live self-migration on a production quality OS such as Linux, while retaining the near-native performance provided by the Xen hypervisor, allows us to minimise the trusted computing base, the privileged code installed on every host in a system. If we can agree on the interface of this software, the rest of the common configuration agreement problem is solved, as users get to supply whatever OS configuration they deem appropriate. Live-migration opens the door to dynamic scheduling, as jobs can now be preempted and reallocated at will. The use of a hypervisor, with a simple interface and good performance and security isolation, shields us from malicious users, and by minimising the trusted computing base, we can take advantage of trusted computing hardware, creating a higher barrier against introduction of malicious hosts into a distributed system.

Future work will focus on the minimised trusted computing base, as well as on security and accounting mechanisms for decentralised systems. Apart from the hypervisor, privileged code must reside on each host for receiving immigrant guest OS's, and from a security viewpoint, a full-blown OS is inappropriate for this task. We aim to implement a minimal host environment, consisting of a TFTP-like service for bootstrapping new domains, as well as cryptographic functionality for signature verification of incoming code, and for interaction with eventual trusted-computing hardware.

6 Conclusion

We have implemented two systems for on-the-fly operating system migration using different approaches. Our major contribution is that such migration is indeed not only doable, but can be done efficiently such that the downtime experienced by an OS is in the millisecond range. Previous systems [15] have focused on reducing the overall time for migration over wide-area links, but not addressed the issue of keeping the OS responsive while migrating.

We also present a novel approach to OS migration, self-migration. While comprehensive benchmarking has not yet been performed, initial results point to OS migration as a viable mechanism for supporting advanced scheduling in clusters and grids, and the novel use of self-migration paves the way for a future secure and minimal trusted computing base definition.

Though there is still work to be done, there are few barriers to the deployment of self-migrating systems in production Grid clusters. Our plans for the next twelve months include trials involving such clusters.

7 Acknowledgements

The original work on NomadBIOS was performed with Asger Jensen (formerly Henriksen). We also owe great thanks to Ian Pratt, Keir Fraser and Steven Hand of Cambridge University, for their work on Xen and co-operation in this project. We would also like to thank Jørgen Sværke Hansen and Peter Andreasen, as well as the anonymous reviewers, for their comments on earlier drafts of this paper.

References

- [1] Fred Douglass and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.
- [2] A. Barak and O. La’adan. The MOSIX multi-computer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.
- [3] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the ninth ACM Symposium on Operating System Principles*, pages 110–119. ACM Press, 1983.
- [4] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.
- [5] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.
- [6] Peter Dömel. Mobile Telescript agents and the Web. In *Digest of Papers. COMPCON ‘96.*, pages 52–57. IEEE Computer Society Press, 1996.
- [7] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master’s thesis, 2002.
- [8] J. Liedtke. On micro-kernel construction. In *Proceedings of the fifteenth ACM Symposium on Operating System Principles*, pages 237–250. ACM Press, 1995.
- [9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of micro-kernel-based systems. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 66–77. ACM Press, 1997.
- [10] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating System Principles*, pages 2–12. ACM Press, 1985.
- [11] C. E. Perkins and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1997.
- [12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM Symposium on Operating System Principles*, pages 164–177. ACM Press, 2003.
- [13] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.
- [14] Espen Skoglund, Christian Ceelen, and Jochen Liedtke. Transparent orthogonal checkpointing through user-level paggers. In *Revised Papers from the 9th International Workshop on Persistent Object Systems*, pages 201–214. Springer-Verlag, 2001.
- [15] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.