CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Single system image: A survey
Authors	Healy, Philip D.;Lynn, Theo;Barrett, Enda;Morrisson, John P.
Publication date	2016-02-17
Original Citation	Healy, P., Lynn, T., Barrett, E. and Morrison, J. P. (2016) 'Single system image: A survey', Journal of Parallel and Distributed Computing, 90-91(Supplement C), pp. 35-51. doi:10.1016/j.jpdc.2016.01.004
Type of publication	Article (peer-reviewed)
Link to publisher's version	10.1016/j.jpdc.2016.01.004
Rights	© 2016 Elsevier Inc. This is the preprint version of an article published in its final form in Journal of Parallel and Distributed Computing, available https://doi.org/10.1016/j.jpdc.2016.01.004. This manuscript version is made available under the CC BY-NC-ND 4.0 licence - https://creativecommons.org/licenses/by-nc-nd/4.0/
Download date	2026-11-04 14:28:00
Item downloaded from	https://hdl.handle.net/10468/4932

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Single System Image: A Survey

Philip Healy[a,b,*], Theo Lynn[a,c], Enda Barrett[a,d], John P. Morrison[a,b]

[a] *Irish Centre for Cloud Computing and Commerce, Dublin City University, Ireland*
[b] *Computer Science Dept., University College Cork, Ireland*
[c] *DCU Business School, Dublin City University, Ireland*
[d] *Software Research Institute, Athlone Institute of Technology, Ireland*

Abstract

Single system image is a computing paradigm where a number of distributed computing resources are aggregated and presented via an interface that maintains the illusion of interaction with a single system. This approach encompasses decades of research using a broad variety of techniques at varying levels of abstraction, from custom hardware and distributed hypervisors through to specialized operating system kernels and user-level tools. Existing classification schemes for SSI technologies are reviewed, and an updated classification scheme is proposed. A survey of implementation techniques is provided along with relevant examples. Notable deployments are examined and insights gained from hands-on experience are summarised. Issues affecting the adoption of kernel-level SSI are identified and discussed in the context of technology adoption literature.

Keywords: Single system image, distributed operating systems, distributed hypervisors, technology adoption.

1. Introduction

The concept of seamlessly aggregating distributed computing resources is an attractive one, as it allows the presentation of a unified view of those resources, both to users and to software at higher layers of abstraction. By hiding the distributed nature of the resources, the effort required to utilize them effectively is diminished. This process of transparent aggregation is said to provide a *single system image* (SSI) of the unified resources.

Introductory surveys of the field have been published in the past by Buyya et al. [1, 2]. However, there have been many developments in the field in the years since their appearance. For example, promising new work has taken place on the implementation of SSI via distributed hypervisors, while kernel-level SSI has not seen widespread adoption despite early enthusiasm and several implementations. Within the broader sphere of

*Corresponding author: Dr Philip Healy, Room 2-22, Western Gateway Building, University College Cork, Western Road, Cork, Ireland. Phone: +353 21 4205935.

Email addresses: p.healy@cs.ucc.ie (Philip Healy), theo.lynn@dcu.ie (Theo Lynn), ebarrett@research.ait.ie (Enda Barrett), j.morrison@cs.ucc.ie (John P. Morrison)

high-performance computing we have seen the rise of multicore systems, GPU acceleration, virtualization and cloud computing. Given these developments, we feel that a retrospective survey of the field, combined with an analysis of its current status and future prospects, is timely.

SSI can be implemented at a number of levels of abstraction from custom hardware and distributed hypervisors through to specialized operating system kernels and user-level tools. At the highest level, message-passing libraries, such as MPI, could be said to provide the application developer with the illusion of working on a multiprocessor architecture even when the underlying execution platform is a loosely-coupled network of workstations. However, the level of transparency presented to the user is inversely proportional to the level of abstraction of the SSI implementation. For example, the illusion of SMP provided by a message-passing library is present only in specially-written application code whereas hardware-level implementations are transparent even to the operating system.

Numerous implementations of SSI have been created over the years at each of the levels of abstraction. A selection of these, organized by abstraction layer, is examined in Section 3. Despite the diverse nature of the various implementations, a number of recurring techniques can be identified. A number of those that appear most frequently in the literature are surveyed in Section 4.

There is much to be learned from those who have gained hands-on experience by deploying SSI technologies to testbeds and production systems. Similarly, there a wealth of information has been published on the topic of performance comparisons and the practical utilization of SSI clusters using a wide variety of application types. These topics are examined in Section 5.

Despite the large body of work expended on the development of numerous freely-available implementations, kernel-level SSI has never emerged as a mainstream cluster computing technique. Many of these implementations are no longer actively maintained at the time of writing, and the academic community seems, to a large extent, to have moved on. On the other hand, promising new developments have taken place in the development of distributed hypervisors that implement SSI (see Section 3.2). Furthermore, the combination of kernel-level SSI implementations with virtualization and cloud computing techniques has opened up new avenues of research. The issue of SSI adoption, or lack thereof, is examined through the lens of relevant literature in Section 6. Our conclusions in light of these observations are presented in Section 7.

2. Definition and Classification

Pfister [3] defines a single system image as *"the illusion that a collection of otherwise independent computing engines is a single computational resource"*. Buyya et al. [2] propose *"the property of a system that hides the heterogeneous and distributed nature of the available resources and presents them to users and applications as a single unified computing resource"*. Lottiaux et al. [4] use the following: *"a physical or logical mechanism giving the illusion that a set of distributed resources (memory, disk or CPU) forms a unique and shared resource"*. Interestingly, Pfister's definition does not stipulate that the resources must be distributed, while the other definitions do.

An important attribute of a single system image implementation is its *transparency*, i.e., the level to which the discrete nature of the underlying resources is hidden from

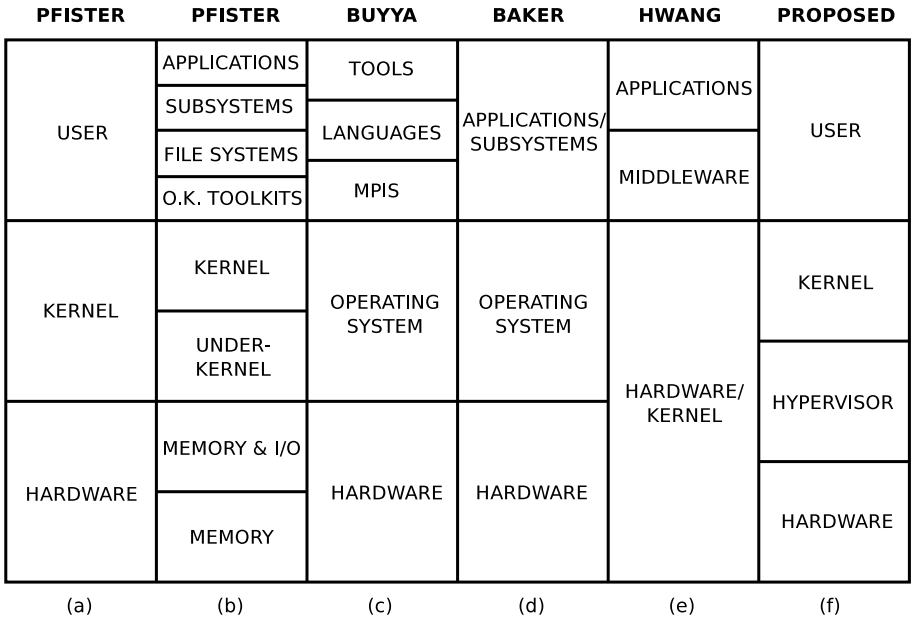


Figure 1: Single system image boundaries and layers as proposed by various authors: (a) Pfister [3] (b) Pfister (detailed) [3] (c) Buyya [1] (d) Baker and Buyya [5] (e) Hwang et al. [6] and (f) our proposed scheme that extends Pfister's through the addition of a hypervisor layer.

the user. Pfister [3] proposes the related concept of a single system image boundary. Boundaries define the levels of abstraction above which aggregated resources appear to be unified, and below which they appear to be separate. Pfister goes on to present a hierarchy of single system image levels, where seven boundary definitions are used to assign SSI implementations to one of eight levels: memory and I/O, memory, under-kernel, kernel, over-kernel toolkit, file system, subsystem and application. The eight levels are grouped into three broad categories: user-space, kernel-level and hardware. Buyya [1] proposes five levels: hardware, operating system, message passing interfaces, language/compiler and tools. A later paper by Baker and Buyya [5] simplifies this to three levels: hardware, operating system and applications/subsystems. Hwang et al. [6] identify three overlapping levels: hardware/kernel, middleware and application software. We will utilize Pfister's three broad categories (hardware-level, kernel-level, user-level) when classifying implementations, with the addition of a fourth: hypervisor-level. Figure 1 depicts the various SSI boundaries and layers that have been proposed.

The transparency provided by single system image is often described in terms of the unification of spaces. These spaces can include:

- *Memory Space* Unification of memory space involves techniques such globally addressable shared memory and process-level distributed shared memory.

- *Process Space* A unified process space can be implemented by providing visibility

and control over the complete set of processes running across all cluster nodes. Techniques such process checkpointing, placement and migration allow for transparent load balancing across nodes.

- *File Space* Distributed file systems can be used to provide an aggregated hierarchy of collections of files stored across a network.
- *Device Space* Unification of I/O devices can provide transparency and/or performance. A cluster-wide virtual character device can be used to provide users with transparent cluster-wide terminal output, while an application that performs extensive internal network I/O may benefit from the ability to harness all of the network cards across a cluster.

Given the broad scope of the techniques that fall within the scope of the classifications above, it is necessary to restrict the range of material covered in this survey. For example, hardware-level techniques, such as distributed shared memory, or user-level techniques, such as message-passing libraries, merit detailed surveys in their own right. A useful filter is to consider whether the authors of relevant literature describe their work in terms of implementing SSI. Furthermore, broadly-applicable techniques such as distributed file systems are adequately covered by existing surveys; we will defer to these as appropriate. We will therefore focus on novel techniques and implementations that are described in terms of SSI and are not adequately covered by more specific surveys. This approach naturally results in an emphasis on the operating system level, which comprises the majority of work in the literature identified by the authors as SSI. However, we will endeavour to highlight seminal and novel contributions at all levels.

3. Implementations

A brief description of each of the most notable SSI implementations, organized by level of abstraction, is presented below. Some of these systems do not fit neatly into a single level of abstraction. For example, kernel-level implementations are often accompanied by a complementary suite of user-level tools. Where ambiguity is present, implementations are assigned to the level of abstraction that best characterises the fundamental approach.

3.1. Hardware-Level

Hardware-level single system image involves the aggregation of discrete computing units into a unified whole that is presented to the operating system. There is inconsistency in the literature as to which technologies should be considered to be hardware SSI. For example, SMP techniques are included in some descriptions [5] but not in others [1]. For our purposes, we will consider hardware systems to provide a single system image only if an element of non-local communication is present; systems built around a local bus, such as classical SMP, are excluded.

Cache coherent non-uniform memory access (ccNUMA) is a distributed shared memory (DSM) architecture where each processor has access to a private memory pool and a global memory pool. Cache coherency ensures that potentially costly cache misses are not incurred when non-local resources are accessed, although a performance penalty for non-local resources is still present.

MIT Alewife was an early implementation of a hardware DSM [7] system where passive backplanes between nodes provide a hardware mesh networking topology to facilitate the distributed shared memory system. Each node is designated a portion of local or private (unshared) memory and can access remote memory via the mesh networking framework. Cache coherence is also supported through a software extended scheme named LimitLess [8].

Additional implementations include the Stanford Dash multiprocessor system [9] implements a ccNUMA style distributed shared memory system. The successor to the Stanford Dash project, the Stanford FLASH multiprocessor [10], addressed the issue of monetary cost through a combination of low-overhead message passing with cache coherent shared memory. The FLASH architecture consists of multiple off-the-shelf processors, each with a separate cache. The system's main memory is distributed amongst the processors. A node controller chip known as MAGIC (Memory And General Interconnect Controller) integrates the memory controller, I/O controller and network interfaces and a programmable protocol processor.

Brock et al. [11] detail a commodity ccNUMA system built from four commodity 4-processor Fujitsu Teamserver SMPs. A cache coherent Fujitsu Synfinity [12] interconnect connects the SMPs together providing the bandwidth necessary to implement DSM across the systems. Through a resource set, applications are able to specify individual processor and memory affinity to improve performance in the ccNUMA system. Similarly, the NUMAchine [13] system, presented by Grbic et al., offers a ccNUMA based hardware DSM implementation using off-the-shelf components. A custom Unix-based operating system, Tornado, was developed to run the machine. A 48 processor DSM prototype demonstrated a peak computational performance of 1.7 Gflops, with the peak bandwidth at any point in the interconnect at 400Mb/s.

A number of commercial ccNUMA machines have been developed such as IBM's NUMA-Q [14], SGI's Origin [15] family of servers and the Unisys ES7000 series [16]. More recently, commercial systems such as the SGI Altix 3000 [17] and HP's Integrity Superdome [18] offer multiprocessor DSMs supporting Intel Itanium processors. Correa et al. [19] implemented a system of multiple memory access levels for their DSM ccNUMA implementation, allowing for varying access overhead global depending relative levels of the communicating processing nodes. As interconnect bandwidth is the limiting factor for DSM systems, commercial systems employ expensive hardware interconnects such as the NUMAlink high speed interconnect offering bandwidth of up to 15 Gb/s [20].

3.2. Hypervisor-Level

There have been several implementations of distributed hypervisors that provide a single guest operating system instance with a single system image of an entire cluster. This approach has the advantage of being largely transparent to the guest OS, eliminating the need for far-reaching kernel modifications and hence allowing for a wider selection of guest operating systems. These systems build a global resource information table of aggregated resources such as memory, interrupts and I/O devices. A virtualized view of these hardware resources is then presented to the guest operating system. Standard distributed shared memory techniques are used to provide a global address space. The single system image is maintained by alternating as necessary between the guest OS and the hypervisor via trap instructions. Note that hypervisor-level virtualization is only one

of several proposed schemes for combining SSI and virtualization technologies; for a more detailed discussion on this topic see Section 4.6.

ScaleMP's vSMP product [21] is, at the time of writing, the only commercially available SSI distributed hypervisor. vSMP provides a single system image of commodity clusters connected via Infiniband. An standard operating runs unmodified above the hypervisor layer, and is presented with a unified view of the cluster as a single SMP machine. Up to 128 nodes can be connected, with a maximum of 1024 processors 256 TB of memory per VM. Benchmarks by [22] indicate that although reading and writing remote memory pages is about 20 times slower than local memory access, significant speedups are still possible, with one OpenMP application achieving a speedup of 80 when run across 104 cores.

Kaneda et al. present the Virtual Multiprocessor [23], a distributed hypervisor for the IA-32 architecture. The hypervisor runs within host OS instances across a cluster and in turn provides a single system image to a guest OS instance via paravirtualization. A testbed system was constructed that used the Virtual Multiprocessor to create an 8-way virtual SMP from eight workstations connected by Gigabit Ethernet. A Fibonacci number calculator was used for performance evaluation. The results showed increased speedup (up to 6.6) for coarse-grained tasks. Finer task granularity results in severely reduced speedup (less than 2).

Chapman and Heiser present the vNUMA system [24], a bare-metal distributed hypervisor for the Itanium architecture that uses paravirtualization of to run an unmodified guest operating system instance at a low privilege level. As the Itanium architecture is not perfectly virtualizable, several sensitive but non-faulting instructions must be replaced by faulting instructions in order to ensure that control passes between guest and host as appropriate. A testbed was constructed from two workstations connected by Gigabit Ethernet. The testbed was then used to conduct a performance analysis using three benchmarks from the SPLASH-2 suite, with good speedups being observed in two cases but poor performance in the third. The overhead resulting from virtualization and kernel paging was found to be low compared to the well-known overhead associated with the distributed shared memory technique.

Peng et al. present the Distributed Virtual Machine Manager (DVMM) [25], a bare-metal distributed hypervisor for the IA-32 architecture that relies on hardware virtualization through the VT-x processor extension. Guest OS instances run unmodified, although kernel-level ccNUMA support is required. The system's reliance on hardware-level functionality implies better performance than Virtual Multiprocessor and vNUMA, although no benchmarks are available to confirm this. Another claimed advantage is the ability to utilize multi-core and/or SMP machines. Song and Xiao[26] describe implementation details of the low-level communications mechanism used to unify the processor, memory, I/O and interrupt spaces. A similar paper by Yong [27] appears to describe the same system. A theoretical model of DVMM (now titled CloudDVMM) and benchmarking data are provided in [28] and [29].

Wang et al. present NEX [30], an extension of the Xen hypervisor that uses a system of distributed co-operating hypervisors to provide a single system image of an SMP machine to unmodified guest operating system instances. This is achieved by integrating a shared memory protocol into Xen's hardware virtualization functionality. QEMU's device model implementation is used to emulate peripheral devices, with shared memory implementations of DMA and MMIO support used to provide a unified I/O space. A

similar Xen-based system is proposed by Ma et al. [31].

MetalSVM [32] is a bare-metal hypervisor for Intel's Single-chip Cloud Computer (SCC) platform. The SCC is composed of 48 P54C cores arranged as a 6×4 mesh of dual-core tiles on a single die. Message passing buffers are provided to allow for high-performance communication between tiles. MetalSVM implements a hypervisor that runs on all SCC cores and unifies the distributed memory spaces into a single shared virtual memory. The hypervisor is implemented using a minimal custom kernel that runs on each SCC core. A standard Linux kernel runs above and is presented with a unified view of the SCC as a single SMP machine. RockyVisor [33] takes a similar approach to SCC virtualization but runs a full Linux kernel on each SCC core. A modified version of the *lguest* kernel module is deployed in each instance, which cooperate to implement a distributed hypervisor. The distributed hypervisor runs a single cross-SCC Linux operating system that is presented with a view of the SCC as a single SMP machine.

3.3. Kernel-Level

The kernel-level approach to single system image encompasses a broad variety of techniques that seek to transparently unify distributed resources. There have been numerous implementations, with the oldest dating back to the 1970s. There are two basic approaches in terms of design philosophy: dedicated distributed operating systems and adaptations of existing operating systems. The latter approach can be further divided into *over-kernel* and *under-kernel* implementations. Under-kernel (or "under-ware" [34]) systems seek to transparently preserve existing APIs, such as POSIX. As a result, existing application code can run without modification. In contrast, over-kernel systems provide additional or extended APIs that allow for efficient utilization of distributed resources. For example, preserving POSIX file pointer semantics for distributed file systems requires a global lock on the pointer; an over-kernel implementation could provide alternative API calls that do not have this requirement. Existing operating systems can be adapted to support single system image semantics in two ways: either by forking the codebase of the OS in question, or by maintaining a patchset that is applied against mainline releases.

Dedicated distributed operating systems are an extensive field of study in their own right. Numerous books and surveys, such as that by Tanenbaum and Van Renesse [35], are available that provide comprehensive overviews of contemporary systems. We will therefore restrict our treatment in Section 3.3.1 to a brief overview of the most notable dedicated distributed operating systems, focusing on single system image aspects. Fork- and patch-based extensions to existing operating systems will be examined individually in subsequent subsections.

3.3.1. Distributed Operating Systems

LOCUS [36] is a Unix-compatible distributed operating system designed and developed by Bruce Walker at UCLA between 1979 and 1983, before being spun off as a commercial entity (Locus Computing Corporation). Single system image functionality is provided via a unified file space, process space and I/O space. File replication is performed automatically. A resolution mechanism is provided that integrates editing conflicts automatically where possible. If resolution cannot be performed automatically then access to the file is disabled and a mail is sent to the user inviting him/her to perform

manual resolution. Transparent remote access is provided to character devices and IPC channels only; raw devices are not aggregated but can be accessed through process migration. Remote execution of programs across heterogeneous CPU architectures (PDP-11 and VAX 750) is supported. A token-based scheme is used to mediate distributed access to resources such as shared memory segments, pipes and file pointers. Some technology from LOCUS eventually made its way, via a circuitous route, into OpenSSI (see Section 3.3.6).

Rozier et al. present Chorus [37], a microkernel-based distributed operating system. Chorus adopts a message-passing approach at the lowest implementation levels through a flexible system of distributed ports and actors. As a consequence, it was possible for O'Connor et al. to add migration functionality using a combination of minimal kernel modifications and a user-space server [38]. The aggressively distributed nature of the operating system allows for transparent unification of the process, I/O and filesystem spaces. A UNIX-compatible version of Chorus, titled Chorus/MiX, was released during the early 1990s.

V [39] is a distributed operating system developed by David Cheriton at Stanford University during the 1980s. Perhaps the most distinguishing feature of V is its unique approach to process migration. By using a process management scheme similar to that developed for the earlier Thoth system [40], multiple V processes can share an address space, making them akin to threads in more conventional systems. However, they can still be migrated independently. A file-oriented device management scheme is provided, allowing distributed processes to access devices transparently as files. Processes that attempt to access device files are suspended until new values are available, simplifying the implementation of processes that perform device I/O. The multi-pass memory copy algorithm used for VM migration in V has since become the standard approach to VM migration, including VMWare's implementation [41].

Ousterhout et al. present Sprite [42], an experimental network operating system developed at the University of California at Berkeley during the late 1980s. Although developed from scratch as a new operating system, the kernel API is similar to that of 4.3 BSD. Network transparency is provided via a fully-transparent network file system and process migration, both of which are implemented through a sophisticated distributed virtual memory system. Caching of remote memory pages is performed automatically in order to improve performance. In order to maintain consistency, the caching of remote files is disabled when they are opened concurrently by another process. Process migration can be performed manually via a shell command or automatically based on the relative loads on the processing nodes.

Amoeba [43] is a network operating system developed by Andrew Tannenbaum and colleagues at the Free University of Amsterdam during the 1980s and early 1990s. Amoeba provides a rich parallel and distributed programming environment, including a custom parallel programming language (Orca). In addition, a UNIX compatibility layer is available. A globally-shared location transparent filesystem is provided. Process placement is supported, but not automatic process migration. However, it is possible to checkpoint a process manually and restart it on a separate machine. A technical comparison of Amoeba with Sprite was performed by Douglis et al. [44].

GENESIS [45] is a distributed operating system that was developed at Deakin University during the late 1990s. Its primary focus is as a platform for parallel applications that provides a single system image and natively supports both the message passing

and distributed shared memory programming models. GENESIS utilizes a client-server microkernel architecture, and the resulting low-level messaging mechanisms are used to efficiently implement the programming model semantics. A global scheduler uses information gathered by a resource discovery manager to efficiently manage cluster-wide process placement and migration. Processes can be created and migrated in groups. Unified file and device spaces are provided, based on their implementation in the earlier RHODOS system [46].

3.3.2. BProc

The Beowulf Distributed Process Space (BProc) [47] is a process migration system implemented by Erik Hendriks as a patchset for the Linux kernel. A master/slave model is used, where processes running on the slaves appearing transparently as local processes on the master. As a result, a single-system image of a cluster-wide process space is presented to users of the master node. Modified kernels are required for both masters and slaves. Migration can only take place via an API call from user code, although this call can take place transparently via a library or command-line tool. Once a process has been migrated, I/O requests are not redirected back to the master, so care must therefore be taken to ensure that shared file systems are used where appropriate. BProc has been successfully used for large deployments, such as the $1,000+$ node Pink and Lightning clusters at Los Alamos (see Section 5).

3.3.3. OSF/1 AD TNC

OSF/1 AD TNC [48] is an extended version of the OSF/1 operating system developed to provide transparent access to multicomputers. This work was performed during the early 1990s as a collaboration between the Open Software Foundation and Locus Computing Corporation and utilises the Mach 3.0 microkernel developed at Carnegie Mellon University. Compatibility is provided with the System V, POSIX and 4.3 BSD APIs. The OSF/1 file system was extended to provide a unified name space, location transparency and remote device handling. A caching mechanism is provided to improve performance when interacting with remote files, and a message-based token protocol is used to synchronize file access. A unified process space and process migration are implemented via a virtual process layer that directs process operations to the node where the physical process is located. System calls are provided for remote forking, migration and execution. Automated load levelling is provided via daemons that implement the load levelling algorithm originally developed for MOS (see Section 3.3.5 below). OSF/1 AD TNC was shipped with the Intel Paragon XP/S Supercomputer and used in deployments of up to 512 nodes.

3.3.4. Solaris MC

Solaris MC [49] is a prototype multi-computer operating system developed at Sun Microsystems Laboratories during the early-to-mid 1990s. It was implemented through modification of the Solaris operating system and provides a number of SSI features while preserving ABI/API compatibility with existing Solaris applications. At the code level, this is achieved by utilising functionality and techniques from the Spring distributed operating system [50] – in particular the use of IDL and CORBA to implement distributed objects using C++. Kernel hooks were added to the source code for Solaris 2.4 and 2.5 in order to interface with the C++ code where appropriate. A global file system,

titled Proxy File System (PXFS), was implemented by extending the existing Solaris file system via the `vnode` interface, allowing the file system to be implemented without kernel modifications. A global process space is provided, with the `/proc` interface extended providing details of all processes in the cluster. Remote execution of processes was implemented, although remote forks and process migration, though planned, were not implemented. A global I/O space is provided that allows cluster-wide access to I/O devices, with the Solaris device naming scheme extended to include the cluster node device names. A packet filter and routing scheme are used to ensure that network connectivity is identical for all applications regardless of cluster node placement.

Solaris MC places a strong emphasis on support for high cluster availability. This is implemented by the cluster membership monitor (CMM), a software component that uses a distributed membership protocol to reach a global agreement on the current cluster configuration. Testing was performed on sixteen dual-processor SPARCstation machines, with parallel makes and multiple copies of a commercial database used to generate workloads. Although Solaris releases were available under an open source license from 2005 to 2010, the source for Solaris MC has not been released to the larger community.

3.3.5. MOSIX and Derivatives

MOSIX (Multicomputer Operating System for Unix, earlier incarnations were referred to simply as MOS) is one of the oldest and perhaps the best-known SSI kernel patchsets. Its origins can be traced back to early work by Amnon Barak on process migration for version 6 of Bell Labs Unix in the late 1970s [51]. Support for version 7 of Bell Labs Unix [52], AT&T Unix System V [53], and Berkeley Unix [54] followed later. From the late 1990s onward development focused exclusively on Linux [55, 56]. MOSIX continues to be actively maintained and developed at the time of writing, and is freely available, albeit under a license that prohibits modification and redistribution.

Process migration in MOSIX is not completely transparent, as a distinction is made between processes that can be migrated and regular Linux processes. Users explicitly specify which processes are suitable for migration by launching them using the `mosrun` command. This distinction is intended to prevent the migration of processes that are unsuitable, either for efficiency reasons or due to their semantics, e.g., `ps`. The `mosrun` command can also be used for process placement, where a normal Linux process is automatically launched on a lightly-loaded remote node.

MOSIX bases automatic process migration decisions on information gathered from running process and the load across the cluster environment. Process metrics include the memory usage, rate of system calls, and the volume of IPC and I/O communications. Cluster node metrics include processing capacity, CPU load, and free memory [57]. A randomized gossip algorithm is used to maintain a distributed bulletin board containing the cluster metrics [58]. Process reassignments can be triggered by a change in the number of processes or their profiles, or a change in the utilization or number of cluster nodes. An opportunity cost algorithm is used to decide whether processes should be migrated and, if so, to which cluster nodes [59].

In the early 2000s it was decided to change the license of future MOSIX releases away from the GPL open source license. As a result, in 2002 a fork of the last GPL release of MOSIX was created by Moshe Bar [60]. The fork, titled *openMosix*, was developed separately from the original MOSIX branch. Improvements to the initial

code base included auto-configuration and node discovery functionality, and new userland tools. Cluster-specific Linux distributions were created that integrated openMosix, such as clusterKnoppix and Chaos [61]. Further development was halted in 2008, when the developers wound up the project [62]. The reason given for the termination of the project was that the increasing popularity of virtualization and multicore processors eliminated much of the market for SSI clustering (see Section 6.3). Development of the openMosix codebase has continued through the creation of another open source project, titled LinuxPMI [63].

3.3.6. OpenSSI and Predecessors

The OpenSSI project, led by Bruce Walker (one of the LOCUS developers), is an attempt to draw together a number of Linux and Unix clustering technologies in order to create an "ideal" cluster operating system [64]. The starting point for the design was the observation that Linux clustering technologies could be grouped into six broad categories: high performance, load-levelling, web service, storage, database and high availability. An ideal cluster operating system would address all of these use cases by improving the availability, scalability and managability of the Linux kernel.

A large part of the implementation originated with code from NonStop Clusters for Unixware project [34], which was contributed by Hewlett-Packard. This code had originated with Locus Computing Corporation but had been further developed in the interim by a number of companies: Platinum, Tandem, Compaq and SCO. During this process, the code base had been ported to AIX, OSF/1 AD, Unix SVR4.x and Unix SVR5. The NonStop Clusters code was used to implement cluster membership, IPC, process management, process migration, clustered filesystems and clusterwide device access. Supported IPC mechanisms include pipes, FIFOs, message queues, semaphores, shared memory and sockets. IBM's distributed lock manager (DLM) [65], implemented as a kernel component and integrated with the cluster membership system, was added as a general-purpose means of maintaining cache coherency. Code from the Linux Virtual Server project (see Section 3.4) was integrated in order to support network-based load leveling. Process-level load-leveling decision algorithms were adapted from MOSIX. User Mode Linux functionality was included as a development aid. Support for a number of cluster filesystem systems was included, including OpenGFS [66] and Lustre [67].

At the time of writing, work on OpenSSI appears to have stalled. The last stable releases (for Fedora Core 3, RHEL 3 and Debian Sarge) were in 2005. The last development release was in 2008.

3.3.7. Kerrighed

Kerrighed [68, 69] is a single-system image implementation for Linux, developed at INRIA during the early-to-mid 2000s. It is implemented as a small (less than 200 lines) patch to the Linux kernel itself along with a number of kernel modules and supporting userland tools. Kerrighed offers the most transparent single system image of all the patchset-based implementations, to the point that the characteristics of individual cluster nodes, such as their load, can be impossible to determine [70]. Much of this transparency results from the use of software abstractions referred to as *containers* and *linkers* [71]. These essentially act as shims between operating system components, implementing a unified cluster-wide address space without requiring modification to the dependant

components. Sequential consistency for distributed memory managed by containers is implemented via a write invalidation protocol.

Uniquely for a Linux-based system, Kerrighed's distributed memory model allows for thread migration, albeit with the attendant inefficiencies caused by OS-managed remote paging (see Section 5). Process migration, process placement and process checkpoint/restart are all supported via a global scheduler. All remote process functionality is implemented using a technique referred to as *process extraction*, where "ghost processes" are used as placeholder for the address space, file handles, signal data and process ID of a remote process. The checkpointing mechanism supports shared memory parallel applications.

Since 2006, the development of Kerrighed has been lead by Kerlabs, a company spun out from INRIA. Various improvements and extensions to Kerrighed were implemented during the late 2000s as part of the XtreemOS research project [72]. The resulting system was titled *LinuxSSI*, and included enhancements to checkpointing, network transparency, scheduling, packaging and testing. The last official Kerrighed release (3.0) was in 2010, although some Ubuntu porting work was carried out by Kerlabs in 2012. However, at the time of writing it is unclear whether or not the project is still under active development.

3.3.8. Clondike

Clondike (CLuster Of Non-Dedicated Inter-operating KErnels) [73] is a Linux patch-set designed to create usable computing clusters from networks of workstations. A client-server model is employed, where a set of permanent core nodes can assign processes to detached nodes, which are workstations available on the local network. The goal is to assign processes to detached nodes when they are idle, and migrate these processes away when user activity resumes. The owners of workstations retain full administrative control over their workstations, and security measures are in place to prevent guest processes from performing potentially malicious actions, such as accessing file system resources outside of the shared cluster filesystem. A kernel patch and kernel modules are used implement the client and server functionality, with NFS used as the shared filesystem. Both process placement and migration are supported. A subset of system calls are forwarded from guest processes to the core nodes, providing partial transparency.

3.4. User-Level

Ghormley et al. present GLUnix (*G*lobal *L*ayer *Unix*) [74], an implementation of SSI on Solaris at the user mode level in the form of a runtime library, command-line tools and an API. Remote execution of unmodified applications is supported, although application support via the API is required to access more advanced features such as parallel execution. Process migration is not supported, and a shared file system across cluster nodes is assumed. The authors concluded that user-mode privileges were insufficient to implement a fully transparent Unix SSI due to issues around terminal I/O, signals and device access, amongst others. This realisation led to the development of SLIC [75], a system that allows for the installation of minimal trusted kernel extensions.

A number of Java Virtual Machine implementations have been created that provide a single system image of a cluster to Java applications. The underlying rationale is that performance gains through parallel execution can be achieved with little effort through the transparent use of distributed processors and memory, resulting in a global thread

space and global memory space. An early implementation was Java/DSM [76], which was built using the existing TreadMarks DSM library. CoJVM [77] and follows a similar approach, but uses a home-based rather than a homeless protocol. JESSICA [78] allows for thread migration rather than thread placement to improve load balancing. DISK [79] differs from the preceding implementations in that it uses Java objects rather than pages as the unit of shared memory. cJVM [80] also uses a shared object model, and supports three remote access methods: method shipping, thread migration and object migration.

Tan et al. present Shell over a Cluster (SHOC) [81], a variant of the Bourne-again (Bash) Unix Shell that provides a single system image of a cluster via a command-line interface. A shared file system across cluster nodes is assumed. A load manager process executes on each cluster node, maintaining a cluster-wide load state by periodically sampling and broadcasting the local load while receiving broadcasts from other nodes. The load manager is consulted whenever a command is entered in order to determine the cluster node on which to launch the resulting process. Furthermore, long-running processes are automatically migrated if a suitable underloaded target node is available. A `forall` extension of the Bash `for` construct is provided to facilitate scatter/gather operations.

Zhang presents Linux Virtual Server (LVS) [82], an implementation of single system image at the network service level. LVS allows scalable and highly available virtual network servers to be created by aggregating clusters of commodity servers. Architecturally, this is achieved by load balancing over a pool of cluster nodes running individual service instances but sharing a common storage backend. The required load balancing functionality was implemented by modifying the TCP/IP stack of the Linux kernel. Initially, this was achieved through patchsets that implemented various techniques (network address translation, IP tunneling and direct routing), although some of this functionality has since been integrated into the mainline Linux kernel. User-space administration and monitoring tools are also provided. LVS was included in Red Hat's Piranha clustering product, and its successor, titled Enterprise Linux Cluster Suite [83].

4. Techniques

A wide variety of implementation techniques have been devised in order to aggregate and unify distributed resources. Work has also been published on security issues and the unification of resources beyond the cluster level by integrating with grids and clouds. As in Section 3 our treatment will reflect the prominence of the kernel-level approach in the literature. Furthermore, we will defer to existing surveys where possible.

4.1. Process Placement and Migration

There are two broad approaches to the provision of unified process spaces: placement and migration [84]. Placement, also referred to as *remote invocation*, is a non-preemptive action that transfers the information required to start the process to another node before execution commences. In contrast, migration is a preemptive action that transfers the state of an already running process to another node. When a process is being migrated, the first step is typically to checkpoint the process, i.e., record the aspects of its state necessary for a later restart [85]. Process state can include attributes such as the register set, memory space and file handles. Some implementations, such as MOSIX, transfer

the memory space completely during the migration phase. Others, such as Kerrighed, migrate memory pages as necessary when they are referenced by the migrated process. The level of transparency of the unified process space also varies. For example, MOSIX differentiates between local processes and those migrated in or out. In contrast, Kerrighed provides a completely unified process space, with the `ps` command listing all processes running across the cluster. Although migration is usually implemented as part of a larger single system image design effort, isolated implementations such as CRAK (a Linux module) [86] have also been developed. A number of previous surveys of process migration tools and techniques have captured the then state of the art, such as those by Smith [87], Nuttall [88] and Milojičić et al. [89]. Table 1 summarises the process management capabilities of the systems surveyed in Section 3.3.

4.2. Distributed File Systems

In the context of SSI, distributed file systems are typically used to provide a unified file system abstraction across nodes. This is achieved by providing location transparency and network transparency. Location transparency is provided if file system paths do not identify the node that the file is physically stored. Network transparency is provided if file operations can be applied equally to local and remote files. Dedicated distributed operating systems, such as LOCUS, have tended to implement custom filesystems to achieve the desired level transparency, while extensions to pre-existing operating systems often leverage existing distributed systems such as NFS. Surveys dealing specifically with distributed file systems (DFSs) have been published by Satyanarayanan [90], and Levy and Silberschatz [91]. These give a good overview of both general purpose DFSs and those developed as part of larger distributed operating system efforts, such as Sprite and Locus. A later survey by Thanh et al. [92] deals with contemporary general purpose systems and proposes a taxonomy.

Early versions of MOSIX used standard NFS [54], although a more advanced MOSIX-specific alternative (DFSA) was developed later that improved performance by integrating with the process migration mechanism [93]. These options were preserved in openMosix [94]. Kerrighed similarly can be configured to use either NFS or KerFS, an experimental system with better performance but reduced stability [95]. OpenSSI can integrate with a number of file systems and distributed storage solutions, such as GFS, OpenGFS, Lustre, OCFS and DRBD [4]. Clondike has been configured to use both NFS and v9FS, a Linux port of the Plan 9 file system [95]. Table 1 includes a column for the file systems available for the systems surveyed in Section 3.3.

4.3. Aggregation of Peripheral Devices

Techniques for presenting unified device spaces range from hardware-level approaches through to APIs. The majority of systems referred to in Section 3.3 implement kernel-level device aggregation to varying degrees of sophistication; the reader is referred to the publications cited there for further details. For the remainder of this section we will focus our attention for the most part on notable non-kernel approaches.

Ho et al. [96] present a system that provides a single I/O space for distributed block storage via a device driver implementation. The device drivers cooperate in a peer-to-peer fashion, with each maintaining a local buffer cache. Non-local I/O requests are redirected to the appropriate peer. A distributed locking scheme is implemented at the

Implementation	Placement	Migration	DSM	Transparency	Extension
Amoeba	Yes	No	No	Complete	No
BProc	Yes	Yes	No	Partial	Yes, Linux
Chorus	No	Yes	No	Complete	No
Clondike	Yes	Yes	No	Partial	Yes, Linux
GENESIS	Yes	Yes	Yes	Complete	No
LOCUS	Yes	Yes	No	Complete	No
Kerrighed	Yes	Yes	Yes	Complete	Yes, Linux
MOSIX	Yes	Yes	No	Partial	Yes, various
OpenSSI	Yes	Yes	No	Complete	Yes, Linux
OSF/1 AD TNC	Yes	Yes	Yes	Complete	Yes, OSF/1
Solaris MC	Yes	No	No	Complete	Yes, Solaris
Sprite	No	Yes	No	Complete	No
V	No	Yes	Yes	Complete	No

Implementation	Filesystems
Amoeba	Custom
BProc	NFS
Chorus	Custom
Clondike	NFS, v9FS
GENESIS	Custom
LOCUS	Custom
Kerrighed	NFS, Custom (KerFS)
MOSIX	NFS, Custom (DFSA)
OpenSSI	GFS, OpenGFS, Lustre, OCFS and DRBD
OSF/1 AD TNC	Custom (UFS, PFS)
Solaris MC	Custom (PXFS)
Sprite	Custom
V	Custom

Table 1: A summary of the capabilities of selected kernel-level SSI implementations: the name of the implementation, whether process placement is supported, whether process migration is supported, whether distributed shared memory across processes is supported, whether the unified process space is fully transparent; whether the implementation is based on an existing operating system; and whether the implementation uses a custom or pre-existing filesystem (shown separately).

buffer cache level in order to maintain consistency. The device driver implementation presents a view of a single large disk to the OS layers above. Petal [97], developed at DEC, is a similar system that uses the Paxos algorithm to maintain consistent global state.

Hypervisor-level SSI implementations (see Section 3.2) typically host a single guest operating system on one of the cluster nodes. Relying on the shared memory implementation to transfer data via Ethernet to another for cluster node for later retransmission via Ethernet is inefficient. An obvious optimization is to transfer data using the locally-available NIC when possible. An extension to the DVMM distributed hypervisor that addresses this issue is presented in [98]. The extension aggregates the NIC resources across the cluster and presents them to the guest OS as a unified virtual NIC while optimizing data transfers in order to reduce unnecessary network traffic. MOSIX implements a similar scheme at the kernel level, allowing migrated processes to communicate directly via sockets rather than routing all communications through home nodes [57].

Virtual OpenCL (VCL) provides an API-level abstraction of the GPUs distributed across a cluster [99]. Allocation is performed through context requests that may specify a number of GPUs. Depending on the size of the request, the VCL runtime environment may allocate one or more GPUs installed in a single machine or a collection of devices spanning several machines. Contexts can be utilized in a transparent fashion using the standard OpenCL programming model. VCL was used as the basis of an extended OpenMP implementation that simplifies the task of writing scatter/gather applications that leverage multiple OpenCL kernels across distributed CPUs and GPUs [100]. A VCL cluster composed of 25 AMD Radeon GPUs was constructed that was capable of computing 25 billion SHA1 hashes per second using the HashCat password recovery algorithm [101].

SGI's Reconfigurable Application-Specific Computing (RASC) devices extend the Altix system infrastructure (see Section 3.1) by directly attaching FPGA coprocessors to the NUMAlink fabric [102]. This arrangement places the FPGAs as peers of the CPUs within the coherency domain of the system, allowing for low-latency, high-bandwidth access. A development environment is provided that supports the use of both high-level (e.g., Handel-C, Mitrion-C) and low-level (e.g., Verilog, VHDL) hardware description languages for generating FPGA configurations. A single system image of the available FPGAs is provided by the RASC Abstraction Layer (RASCAL), a software stack that performs automatic wide scaling across multiple FPGAs [103].

4.4. Security Issues

Most kernel-level SSI implementations assume a trusted network where only authorized nodes are allowed, with security measures in place at the cluster perimeter. In practise, this would typically be enforced using VLANs and/or subnets. Security warnings about the dangers of untrusted networks are to be found in the manuals for the leading systems: BProc, MOSIX, openMosix and Kerrighed. These systems are not hardened against malicious network packets, leaving them vulnerable to denial-of-service attacks. A National Vulnerability Database entry (CVE-2002-2079) for MOSIX and openMosix was created to this effect in 2002. The MOSIX manual [57] lists a number of potential security vulnerabilities that should be guarded against: theft of super-user rights, IP masquerading by hostile entities, leaking of multi-cluster passwords, and unauthorized job migration. The latter issue is addressed in MOSIX through the use of client

and server keys; servers will only accept jobs if their key matches that presented by the client.

Latter [104] considers the security issues that arise when kernel-level SSI systems are deployed across wide area networks, such as multi-site corporate networks or the Internet, rather than in restricted subnets. A security analysis of openMosix is performed and a number of problematic areas are identified: node discovery relies on multicast, no authentication mechanism is in place to prevent malicious nodes from joining clusters, and communications between nodes are neither encrypted nor validated. A number of improvements and best practices are suggested: the addition and removal of nodes to the cluster should be securely negotiated, IPSEC should be used to create secure inter-node VPN tunnels, and packet filtering should be used to exclude external traffic from vulnerable ports. Additionally, the Layer-2 Tunneling Protocol could be used to provide a virtual Class B network inside the IPSEC tunnel mesh between cluster nodes. The performance implications of the proposed security enhancements are not considered.

Kačer and Tvrdík [105] examine security issues around process migration, where processes that have access to sensitive information can potentially be compromised after migration to malicious nodes. Potential threats are identified: running processes may be modified so that unauthorized file access is attempted on the home node; processes may be migrated with sensitive information in their memory space; and, processes may be modified before being migrated elsewhere. A process labeling method, referred to as *stigmata* is proposed that addresses these concerns. Performing a sensitive actions results in a process being permanently labeled with an appropriate stigma. Any stigmata assigned to a process are inherited by its children. The presence of a stigma can prevent a process from being migrated to an untrusted node. Attempting to perform a sensitive action while migrated results in the guest process being terminated and its state restored from a checkpoint on a node with appropriate permissions.

Štava and Tvrdík [95] consider issues around filesystem security that arise when SSI systems are deployed to non-dedicated clusters. Authentication and privacy issues are addressed, in particular the enforcement of access control to sensitive files in the absence of administrative control over all cluster nodes. The Clondike kernel-level SSI implementation in conjunction with the v9fs filesystem from the Plan 9 operating was used as the starting point. The OpenSSL library was used to augment v9fs with authentication and encryption mechanisms. Access control was implemented using a configuration mechanism that allows access to files and directories to be restricted to trusted nodes. Migrated processes are assigned private filesystem namespaces with the privileges of the process owner. A performance evaluation was performed, and the authors note an almost 20-fold increase in mounting time when authentication is enabled. Furthermore, a significant throughput decrease of up to 50% was observed when encryption is enabled.

4.5. Grid Integration

MOSIX has been extended with a grid management system that enables the creation of federated clusters-of-clusters [106]. Automatic resource discovery is implemented via a randomized gossip algorithm. Inter-cluster process migration is supported, with LZOP compression of process memory used to minimize transfer times. A sandboxing scheme, based on the interception of system calls, is used to ensure that migrated processes cannot access local resources. The system adapts automatically to the addition and removal of new clusters. Users can control which of their cluster machines are available for use by

other grid members via the creation of cluster partitions. Local processes take precedence over those of guests, and a flood control system is in place to prevent local machines from becoming overloaded with guest processes. An economy-aware extension was developed that allows inter-cluster process migrations to be scheduled via a spot market [107].

XtreemOS [108] is an operating system that transparently utilizes grid resources at the kernel level. Secure resource management is provided through the implementation of virtual organizations. One of the three available flavors of the system integrates with the LinuxSSI patchset, enabling access to federated grid resources on single system image clusters (see Section 3.3.6). Transparent integration of out-of-cluster compute resources into SSI clusters is not supported. However, the implementation of checkpointing and process group migration features allow jobs to be migrated from one cluster to another [109].

4.6. Virtualization and Cloud Integration

SSI and virtualization are closely-related techniques in that they both abstract resources on behalf of the user. SSI implementations allow a collection of discrete physical machines to be presented to the user in the guise of a single virtual machine. Similarly, virtualization hypervisors and IaaS implementations allow physical machines to be presented to the user in the guise of a collection of discrete virtual machines. From the user's point of view, a collection of machines exposed via SSI is effectively aggregated into a logical whole, while the same machines exposed via virtulization and IaaS would appear as a diverse collection of discrete entities. However, the approaches ar somewhat similar in that they divorce the resources available to an OS from the physical hardware.

The earliest work on virtualization was performed by IBM in the 1960s, with systems such as the CP-67 and VM/370 pioneering providing the first widely adopted implementations [110]. The theoretical foundations for virtualization were developed by Goldberg, Popek and others in the late 1960s and early 1970s [111]. These were developed further by Robin [112] into the common classification of hypervisors into three types, where Type 1 hypervisors that run directly on the physical hardware, Type 2 hypervisors that are hosted within a conventional operating system environment, and hybrid hypervisors that execute privileged instructions in software. This interpretation of Goldberg's work has been challenged, however, with some commentators arguing that practically all hypervisors are technically Type 1 according to Goldberg's original definition [113]. Furthermore, container-based virtualization, a lighter-weight alternative to full virtualization, has since become popular. Container-based schemes avoid running a full hypervisor by using the same operating system instance for both the host and guests. Implementations include LXC (Linux), Zones (Solaris) and Jails (FreeBSD) [114].

Possible combinations of SSI with virtualization technology are examined by Gallard et al. [115]: Type 1 virtualization via SSI and *vice versa*, Type 2 virtualization via SSI and *vice versa*, and containers on SSI and *vice versa*. Each scenario is evaluated in terms of what the authors identify as the advantageous capabilities of virtualization: isolation, server consolidation, application portability, virtual machine portability and suspend/restart. Examples of SSI on Type 1 and 2 virtualization are given in Section 3.2 above. More complex arrangements in the form of multi-level arrangements of virtualization and SSI are proposed in a follow-on paper [116]. In these architectures, portability is delivered by virtualization while resource aggregation is provided by SSI. This layer-

ing would allow applications written for a given processor to take advantage of all the resources in a cluster even if the cluster uses another processor architecture.

Maoz et al. [117] present a system that extends MOSIX by allowing processes to be launched in virtual machines, which can themselves be migrated. A virtual machine that encapsulates a job is created automatically on submission. A simple inter-cluster file sharing scheme is supported through the specification of a list of files to be packed into the VM. A pool of cached VM instances can be used to reduce job startup time. Redirection of the VM I/O streams and signals back to the launch node is performed automatically. Two migration modes are supported: *fold*, where running processes are migrated back to the home-node VM before the VM itself is migrated, and *no-fold*, where the home-node VM and its distributed processes are migrated separately. The no-fold mode was found to have better performance, but requires the MOSIX to be running at both migration endpoints.

The on-demand nature of the cloud paradigm enables various techniques for scaling applications at the VM, network and platform levels [118]. Tools that simplify the creation of scalable cloud-based compute clusters by bridging the gap between IaaS and PaaS are increasingly available. For example, the Nimbus project has developed a suite of tools that simplify the execution of scientific computing applications in the cloud by automating the tasks of instantiating, configuring, monitoring, repairing and scaling IaaS clusters [119]. Creating an SSI instance from a pool of virtual resources provisioned from an IaaS provider affords the ability to leverage the "on-demand" nature of IaaS to quickly and easily adjust the size of the resource pool.

A number of scenarios that combine the XtreemOS grid operating system (see Section 4.5) with cloud technologies are proposed by Morin et al. [120]. One of these is the creation of a large virtual SMP machine using the SSI flavour of XtreemOS. This scenario is proposed as a means of creating on-demand SMP virtual machine instances incorporating much more processing power than the CPU maximum typically offered by IaaS providers. The creation of XtreemOS SSI clusters from compute resources federated across multiple IaaS providers is proposed by Kielmann et al. [121].

ElasticSSI [122] is a proposal to implement on-demand provisioning of SSI clusters via a PaaS interface. The number of virtual machines that comprise individual clusters would be adjusted through the application of elastic scaling. The automation of the scaling process would result in clusters that are self-optimizing with respect to resource utilization; virtual resources would be allocated and released based the value of system load metrics, which in turn would be dependent on the resources allocated. The scaling process would be transparent to the end user as the SSI implementation would maintain the illusion of interacting with a single OS instance. Heterogeneity could be introduced into the resource pool by allocating virtual machine instances of varying types based on the value of metrics such as system-wide CPU load and memory utilization. Pfister [123] proposed a similar scheme for combining multiple multicore virtual machines provisioned via IaaS.

The factored operating system (*fos*) [124], developed at MIT, provides a single system image across both multicore machines and cloud virtual machines. This characteristic feature of this approach is that the operating system is factored into function-specific services, where each service is distributed into spatially distributed servers. Operating system specific functions such as file system access and physical memory allocation are performed by a combination of one or more servers known collectively as a *fleet*. Each

server resides on a single core with applications running on separate cores. The system is implemented as a paravirtualized machine in order to facilitate deployment to public clouds.

MOSIX Reach the Clouds (MOSRC) is an extension to MOSIX that allows migrated processes to access files without copying them to the remote execution environment [57]. This allows applications to leverage commercial clouds to process locally-stored data or *vice versa*. MOSRC is intended for situations where organizations do not wish to store data with commercial cloud providers, or wish to conveniently process data that is stored remotely.

4.7. Miscellaneous

Scyld ClusterWare [125] is a cluster management software suite that provides a single system image of a cluster at the configuration level. Changes to the configuration on the master are pushed out automatically to compute nodes, maintaining a consistent cluster-wide operating environment. By providing a central point of configuration, compute nodes can be added, updated and reprovisioned easily. Support for distributed process spaces using BProc is also included.

Plurix [126] (not to be confused with the Brazilian operating system of the same name) is a distributed shared memory operating system implemented using Java. A custom compiler is used to provide access to the low-level hardware resources, such as device registers, that are not normally available to Java applications. The memory available in cluster nodes is organized into Distributed Heap Storage (DHS) containing both data and code. The DHS is page-based, allowing hardware MMU functionality to leveraged in order to efficiently trigger network transfers of heap segments. A token-based transactional consistency model is used, ensuring that only one node at a time enters the commit phase. Automatic process migration is performed via a load balancing mechanism. Details of the application development process are vague, but it appears to be based on standard Java code.

5. Deployment, Utilization and Benchmarking

Brock et al. [11] describe a 16-way ccNUMA system constructed at IBM from four individual four-way SMP machines. A Synfinity switch was used to provide the cache-coherent interconnect. The system featured a total 16 350MHz Intel Xeon processors and 4GB of RAM. A performance monitoring card was used to observe traffic on the interconnect, allowing remote memory accesses to be monitored without affecting performance. A set of APIs that provided a resource set abstraction was provided in order to allows applications fine-grained control over memory and processor affinity. The performance of six applications from the Splash-2 benchmark were evaluated. The authors concluded that the results were "not ideal" but more than adequate for several applications.

A 1024-node SSI cluster, titled Pink, was deployed at Los Alamos National Labs in 2003 [127]. At the time, it was the world's largest single system image cluster. A Myrinet interconnect was used to provide high-speed network connectivity between the 1024 dual-processor 2.4 GHz Intel Xeon compute nodes, each containing 2GB of memory. The resulting system had a theoretical performance of 9.6 Teraflops, with a nominal

price/performance ratio of 0.625 USD per Megaflop. BProc was used to provide single system image at the kernel level. Pink was built as a proof-of-concept testbed for Lightning, a similar 1408-node cluster later deployed to production.

Kofahi et al. [128] evaluate the performance of a MOSIX deployment on a six-node Linux cluster with a Fast Ethernet interconnect. Execution times were recorded for applications with two implementations: one using migratable processes and another using LAM MPI. The rationale of this approach is that MOSIX's load balancing functionality might result in better efficiency than a message passing approach. The applications considered were an unnamed CPU-bound computation (apparently busy waiting) both with and without background load, and a matrix multiplication application. In all cases the MOSIX approach was found to perform better, with speedups ranging from negligible values to greater than 1.5.

A comparison of openMosix, OpenSSI and Kerrighed was performed by Lottiaux et al. [4]. The feature sets of the three systems are compared in a number of areas, including transparency, process management, IPC management, fault tolerance and hardware support. A comparative performance evaluation is provided for a number of common features: process migration, stream migration and file system bandwidth. A limited comparison of shared memory performance was performed. This activity was limited by the fact that only Kerrighed has native shared memory support; OpenSSI provides limited support for System V shared memory segments, while openMosix has no built-in support for shared memory. Only Kerrighed exhibited a speedup for benchmark applications that utilize System V shared memory. The authors conclude that, at the time of writing, OpenSSI was the most stable of the three systems while Kerrighed offered the best performance.

Osiński and Niewiadomska-Szynkiewicz [70] performed a similar three-way comparison between openMosix, OpenSSI and Kerrighed. Evaluation was performed on a heterogeneous three-node cluster. Two applications were considered: a Monte Carlo simulation and a an application comprised of a series of system calls. The results mirrored those of Lottiuax: Kerrighed was found to be the best-performing system in terms of speedup but was not found to be stable; attempting to run the Monte Carlo application with large numbers of processes resulted in cluster-wide system crashes. Interestingly, detailed load balancing results for Kerrighed could not be obtained due to the extent of the transparency it provides, i.e., it was not possible to determine the load on individual cluster nodes. OpenSSI performed best in terms network bandwidth utilization for the Monte Carlo simulation, but was again surpassed by Kerrighed during the system calls benchmark.

A library, titled *gthreads*, allows OpenMP applications to run unmodified across Kerrighed clusters. This is achieved by providing an implementation of the POSIX threads interface that hooks into Kerrighed modules [129]. In effect, the entire cluster is exposed to the OpenMP runtime environment as a single SMP machine. An optimization is provided that avoids the maintenance of cross-cluster memory coherency for thread-private data. Implementation of POSIX thread synchronization primitives, such as locks and semaphores, is delegated to the corresponding Kerrighed module. Performance evaluation was performed by running a water flow modeling application of four-node cluster comprising four compute nodes, each with 512 MB of memory and a single Pentium III 500MHz processor, connected by a 100Mb Ethernet network. Evaluation of the application's performance on a true four-way SMP machine determined a speedup of 3.62. In

contrast, the speedup on the virtual SMP machine was 0.46, i.e., a slowdown of greater than two. The authors conclude that this approach *"won't result in efficient code when executed on a SSI operating system"* and that *"specific work should be done to take into account the specifics of SSI operating systems."* Nevertheless, results obtained on more modern hardware would be of interest. The authors also reference optimization techniques that could be used to improve performance [130].

6. Adoption of Kernel-level SSI

Kernel-level SSI seemed to be on the cusp of emerging as a mainstream technique during late 1990s, and features prominently in introductory books on cluster computing from this period [131, 132, 5]. Pfister's 1998 book [131] devoted an entire chapter to the benefits of kernel-level SSI. Just over a decade later, he posed the following question in his blog: *"Why hasn't SSI taken over the world?"* He went on to highlight the lack of adoption of kernel-level SSI as follows: *"After last-minute... pull-outs and half-hearted product introductions by so many major industry players - IBM, HP, Sun, Tandem, Intel, SCO - you have to ask what's wrong."* There is an established and rich literature on the adoption of technology that may provide some answers. While a deep discussion of technology adoption literature is outside the scope of this article, extant research may provide a lens through which we may begin to understand the failure of kernel-level SSI to be adopted widely.

This section provides an overview of the claimed benefits and drawbacks associated with kernel-level SSI, and examines the extent to which virtualization technology competes against it. Finally, key technology adoption literature is used to analyse the failure of kernel-level SSI to be adopted as a mainstream technology. Three specific perspectives are considered: diffusion of innovation, standards of economics and disruptive innovation.

Most modern clusters use freely-available open source operating systems, with Linux being the most popular. As a result, cluster administrators are free to choose their preferred operating system without regard to cost. As several SSI implementations, such as Kerrighed, are themselves freely available, the issue of selling SSI in a commercial sense does not arise; the choice of whether or not to adopt it depends largely on utility rather than cost.

An assessment of the lack of adoption of process migration, which could be considered a sub-field of kernel-level SSI, was performed by Milojičić et al. [89]. The authors identify a number of misconceptions that they believe are widely held: that implementations of process migration are complex, entail unacceptable costs, lack support for transparency, and lack support for heterogeneity. In contrast, they identify what they regard as the true barriers to adoption: lack of applications, lack of infrastructure, lack of necessity, and sociological factors. These are themes that arise again during our discussion below, except applied more broadly to kernel-level SSI as a whole.

6.1. Claimed Benefits

Kernel-level SSI seeks to diminish the effort required to utilize distributed computing resources through transparent aggregation. Claimed benefits typically center on the ability of transparency to simplify the administration and use of clusters. Administrators can take advantage of transparency to manage the cluster-wide OS as a single unit.

Similarly, features such as distributed file systems and process migration allow users to use familiar tools and abstractions in a transparent fashion while working with distributed resources. More specifically, Buyya et al. [1, 2] identify the following benefits of SSI:

- A simple, straightforward view of all system resources and activities from any node in the cluster.
- Users do not need to concern themselves with where in the cluster their applications will run.
- Resources can be used transparently irrespective of their physical location.
- Users can work with familiar interfaces and commands.
- Administrators can manage the entire cluster as a single entity.
- Reduced risk of operator errors, resulting in improved performance, reliability, and higher availability.
- Avoids the need for skilled administrators.
- Simplifies system management and thus reduces cost of ownership.

6.2. Drawbacks

The literature and anecdotal evidence suggest a number of drawbacks that hinder the deployment of SSI clusters in practise. These include:

- *Performance and Scalability* Most kernel-level SSI implementations attempt to mimic the behaviour of familiar single-node operating systems as closely as possible. Unsurprisingly, many single-node OS features, such as file position pointers, were not designed with horizontal scalability in mind. Distributed implementations of these features necessitate the implementation of system-wide shared resources protected by locks, leading to severe contention issues at scale. This leads to a conflict between the desire to accurately mimic the features provided by single-node OSes and the inability of those features to scale [123].
- *Security* As discussed earlier in Section 4.4, most kernel-level SSI implementations assume a trusted network where only authorised nodes are allowed, with security measures in place at the cluster perimeter. SSI implementations are vulnerable to a number of potential security vulnerabilities that are likely to be unacceptable to enterprise IT departments.
- *Concurrent Application Instances* Many applications have been designed and implemented under the assumption that they will execute in a unary fashion on a single OS instance at a time. One of the reasons for the success of virtualization is that the isolation provided by virtual machines allows many of these of these unary applications to be consolidated on a single physical machine. Conversely, a single system image implementation would be forced to either modify the application or else run a single instance irrespective of the number of cluster nodes. For example, a scalable web serving cluster might utilize multiple instances of the Apache

web server running on multiple physical or virtual machines, which are clustered together to load balance incoming requests. Replicating this arrangement using process migration would be difficult, as a host of issues, such as contention for ports and files, would arise if multiple instances of Apache were run simultaneously [123].

- *Cluster Availability* Ease of management is often cited as a benefit of SSI operating systems. However, occasionally the operating system itself must be upgraded. The favoured approach for updating clusters is to roll out updates to one node at a time. This staggered approach to updating ensures that the cluster maintains high availability during the update process. Newly-updated nodes are typically evaluated thoroughly and tested for problems before subsequent nodes are examined. This is a time consuming but proven approach. SSI implementations can simplify this by simultaneously updating the entire cluster. However, under this scenario a single faulty update can potentially bring the entire cluster offline. This is a risk that many cluster administrators do not appear to be willing to take [123].

- *Application Placement* Distributed systems are often carefully managed by expert system administrators who place applications on specific nodes in order to maximise overall system performance and reduce interference among competing processes [133, 134]. One of the advantages of SSI is that processes are migrated between nodes in a seamless manner. However, this feature can be a double-edged sword as automated migrations may result in placements that are obviously suboptimal to a human operator. For example, a number of I/O intensive processes may be migrated to the same cluster node, resulting in the locally-available I/O bandwidth being saturated.

- *Installation Conflicts* Most contemporary kernel-level SSI implementations are implemented as patchsets against individual kernels. The installation process involves downloading the kernel source, applying the patchset then compiling and installing the new kernel along with required support utilities. Cluster-level configuration may also be required if auto-discovery is not supported. Many Linux distributions ship with modified versions of mainline kernel releases. This can potentially lead to conflicts between the distribution and SSI patchsets.

Some of the issues outlined above can be solved or mitigated. Modern configuration management tools (such as CFEngine, Chef and Puppet) simplify the task of updating cluster nodes. Issues with multiple instances of the same application can be addressed with container migration (see Section 4.6). Control over application placement could be implemented through the provision of suitable support tools. Installation and support issues can be resolved given the appropriate level of resourcing – an easy to install MOSIX virtual machine image is available, and commercial support for Kerrighed is advertised by Kerlabs.

Other issues, such as the limitations of distributed shared memory, are fundamental and intractable given the current state of computing technology. Given the well-known fallacies of distributed computing [135] it could be argued that the transparency provided by SSI implementations will often be a "leaky abstraction" [136]. This is particularly true for the majority of clusters that rely on commodity Ethernet networks rather than

high-performance cache-coherent interconnects such as SGI's Altix [17]. Whether kernel-level SSI implementations are truly suitable for use as general-purpose cluster operating systems is therefore open to debate.

Furthermore, SSI operating systems are composed of separable concepts and components, each of which has competition and/or is potentially more useful when used in isolation. Configuration management tools such Puppet and Chef provide centralised administration. Distributed file systems such as NFS and Lustre are widely used in non-SSI clusters. Distributed process and job management can be achieved using queue managers, such as PBS, rather than process migration. Fine-grained parallelism can be achieved through the use of programming libraries such MPI and openMP. Increasingly, users interact with clusters through higher-level platforms such as Apache Spark. Although these views of the system are far from transparent, transparency, as noted above, brings its own issues.

6.3. Competition from Virtualization

As noted in Section 4.6, SSI and virtualization are orthogonal approaches in some respects as the former involves the aggregation of resources in a unified whole, while the latter involves the division of resources into discrete virtual machines. From an administration point of view, however, both approaches can be very similar: distributed virtualization systems, such as VMware ESX, aggregate distributed computing resources and present them as a single, unified, whole with a centralized point of administration [137]. Tellingly, this transparency is presented to administrators as a management aid rather than to end users in the form of additional operating system functionality. Migration is supported, but at the VM rather than the process level [41].

From the users' point of view, discrete OS instances afford heterogeneity, familiarity and isolation. Application code can be run unmodified. Parallelism can be supported by running standard parallel processing frameworks across collections of virtual machines. It could therefore be argued that virtualization has, to a large extent, usurped the role intended for SSI operating systems. Under this analysis, the advocates for SSI were correct in their assessment of the value of techniques such as aggregation and migration but misjudged the level of abstraction at which this functionality is most useful: at the VM or container level rather than the process level.

That the ascendancy of virtualization coincided with the decline of SSI was not lost on researchers in the field. In the tellingly-titled report *"Is Virtualization Killing Single System Image Research?"*, Gallard et al. [116] point to the rise of virtualization technology for the decline of SSI, which they identify as providing competing capabilities. The abandonment of the openMosix project cited as evidence. The members of the openMosix project themselves gave the following rationale for its discontinuation [62]: *"The increasing power and availability of low cost multi-core processors is rapidly making single-system image (SSI) clustering less of a factor in computing. The direction of computing is clear and key developers are moving into newer virtualization approaches and other projects."*

6.4. Analysis

Technology adoption literature focuses considerably on the process of technology diffusion and the factors influencing technology adoption decisions. Rogers [138] suggests

five characteristics that influence the adoption decision of any given technology: relative advantage, compatibility, complexity, trialability and observability. Depietro et al. [139] extended the adoption debate by recognising that in addition to those characteristics identified by Rogers, characteristics of the adopting organisation such as communication, and control and the environment in which the organisation operates, also play an important role. In the case of interorganizational systems, Iacovou et al. [140] suggest that perceived benefits, organizational readiness, and external pressure play a role in adoption. External pressure is generated from competitive factors and the relative power of trading partners.

Research has found that early adopters are more likely to be large firms [141, 142], firms for whom an innovation is more likely to be most profitable [141, 143], and firms with strong competitive positions [144]. Eveland and Tornatzky [145] list five elements of context that influence technology adoption and diffusion, namely: (i) scientific complexity, (ii) technological fragility, (iii) level of post-sale support, (iv) organisational impact, and (v) the ease of productization. Similarly, research by Davis et al. [146] found that perceived usefulness and ease of use were the most significant determinants of technology adoption; recent research suggests that interorganizational trust has a significant positive influence on these factors [147].

Another strand of technology literature focuses on the economic costs and benefits of the technology adoption including both switching costs and network effects [148, 149]. Fichman and Kemerer [150] argue that software innovations are more likely to become dominant technologies when they score highly on both diffusion of innovation and economics of standards criteria. They also suggest that expectations about the technology's chances of dominance with positive expectations being reinforced by a strong scientific base and a clear match between the technology's unique strengths and industry trend. Recent research on technology adoption has seen the emergence of more integrative studies that bring these various perspectives together [151].

Dedrick and West [152] suggest that a distinction needs to be made between the adoption of an innovation and an adoption of a variant of the same fundamental technology. Bower and Christensen [153] classified innovations as either sustaining or disruptive with the former being the more prevalent. Sustaining innovations typically involve the evolution of existing products and services by adding value or functionality; by contrast, disruptive innovation creates new value networks and therefore impacts entire markets [154]. Rigby et al. [155] propose three tests for establishing the disruptiveness of a given innovation:

1. Does the innovation target customers who in the past haven't been able to "do it themselves" for lack of money or skills?
2. Is the innovation aimed at customers who will welcome a simple product?
3. Will the innovation help customers do more easily and effectively what they are already trying to do? Christensen [154] notes that for an innovation to be disruptive, the incumbent product and service offerings must overserve the existing market needs.

6.4.1. Diffusion of Innovation Perspective

As noted in Section 6.1, the claimed benefits of kernel-level SSI center on ease of use and ease of administration. However, there is little empirical evidence that SSI is more

effective in diminishing the utilization effort of distributed computing resources in relative terms (e.g., in the case of virtualization) or for generalised use cases. Furthermore, the drawbacks identified in Section 6.2 may actively work against its adoption.

Pfister [123] reported on his experience of two IBM focus groups on SSI. One group included IBM customers running departmental servers with little knowledge of clusters; the other with operational knowledge of running clusters. The former found the presentation on SSI too complex - they could neither understand what SSI was nor why someone would need it. The latter *"were amazed that it was possible, and thought it was really neat, although some expressed doubt that anybody could actually make it work."* In both focus groups, the IBM customers would not adopt SSI. The academic literature and this anecdotal evidence suggests that for most enterprise IT departments SSI would be a complex technology to adopt. This complexity impacts the likelihood of adoption in a number of ways. It requires a new skillset and a change to operational practices; both of which require an upfront training investment. Furthermore, maintenance and support for SSI installations is likely to suffer from the limited number of SSI specialists and both a relatively small experience base and SSI professional community. This complexity also has knock-on effects on trialability and observability. Even limited SSI trials require a significant human resource and system investment and the key impact, efficiency in utilising distributed computing resources through transparent aggregation is difficult to observe by definition. SSI as a technology is difficult for peers to observe in that it is used on systems that perform a backend rather than public-facing role.

6.4.2. Standards of Economics Perspective

It would appear from this analysis that SSI did not have the necessary characteristics suggested by Rogers [138]. An analysis from the perspective of the economics of standards literature also provides insights in to SSIs failure to evolve in to a mainstream technology. SSI failed to achieve increasing returns of adoption from those sources critical to software adoption, namely learning by using, positive network externalities and technological inter-relatednesss [150]. Firstly, the relatively small community of developers working on SSI resulted in a slow accumulation of experience in developing and applying the technology. Significant technical shortfalls remained and a base of experience developers did not evolve. Secondly, SSI, as a technology, did not scale sufficiently even at an early stage to benefit strongly from positive network effects. Given the niche use scenarios, it is unlikely whether network effects would be strong even if adoption increased. Thirdly, SSI did not have a large base of compatible products and services. In fact, as discussed earlier SSI was both incompatible with other software products and operational practices. Furthermore, it is clear from the comments in IBM focus groups and the literature that SSI also suffered from a strong prior technology drag. The installed base of prior technologies including operating systems, applications and associated programming models retarded adoption.

SSI did have a number of large, financially stable and influential sponsors; IBM, Sun Microsystems, Intel, Tandem, SCO and SGI, amongst others, explored the technical and commercial feasibility of SSI. Notwithstanding this, it is clear from Pfister [3] that, at least in the case of IBM, these sponsors found it difficult to identify a commercial market for SSI; customer expectations were low. A decline in interest in SSI within the academic community is also evident from the reduction of publication activity observed

by employing the n-gram frequency analysis technique from the field of Culturomics [156] (see Figure 2).

Despite the presence of strong sponsors, the combined effect of these factors is that SSI may have suffered from a reluctance of these sponsors to further commercialise SSI from what Farrell and Saloner [157] refer to as symmetric inertia. Symmetric inertia arises when firms only moderately favour a change to a new technology and therefore are not individually sufficiently motivated to drive the technology forward. In the case of SSI, the factors would appear to have retarded the development of the SSI community and network to a critical mass and introduced sponsor fear of a risk of stranding caused by "backing the wrong horse". The resulting excess inertia could explain SSI adoption failure.

6.4.3. Disruptive Innovation Perspective

SSI also fails to meet the tests laid out by Christensen [154]. Firstly, SSI did not target customers who in the past haven't been able to "do it themselves" for lack of money or skills. SSI remained relatively costly and required specialised knowledge. Similarly, SSI failed the second test in that it was not simpler but rather more complex to execute than existing solutions. Thirdly, while SSI might deliver the same or better performance, it was not perceived to be easier to execute. Incumbent product and service offerings were not over-serving the existing market needs.

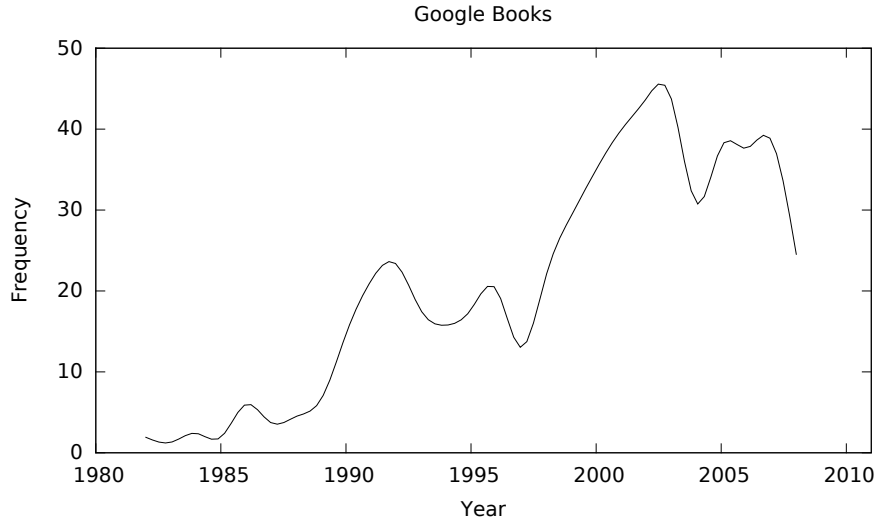
From the analysis above, it would appear SSI did not score highly on criteria relating to diffusion of innovation criteria, economics of standards or disruptive innovation. To paraphrase Eveland and Tornatzky [145], SSI was scientifically complex, technologically fragile, required specialized rare post-sale support, had low organisational impact, and was not easy to productise.

7. Conclusion

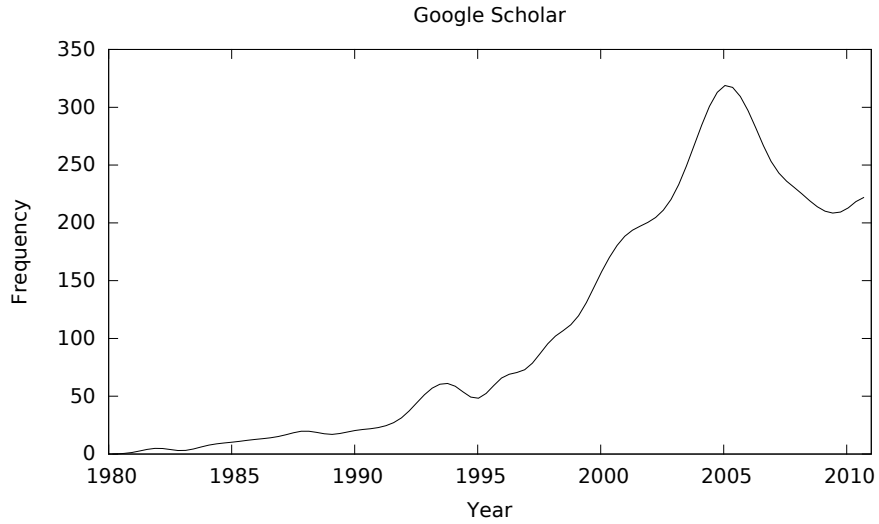
Single system image embodies a rich variety of techniques and implementations with a history going back over three decades. Many systems were developed in academic or industrial contexts and are no longer available. Nevertheless, details of their implementation have been published in the literature, resulting in a significant body of work that can be exploited in future SSI implementations and related areas such as distributed file systems. Other implementations are available under open source licenses, allowing development to continue into the future given sufficient interest.

Notwithstanding some notable exceptions, such as the as the inclusion of BProc in the Scyld ClusterWare product, kernel-level SSI has gained little traction in the marketplace. The distributed hypervisor approach to SSI is an exciting development but remains immature. The rise of virtualization has been cited in some quarters as the primary cause of the demise of kernel-level SSI. We believe that this view is over-simplistic. Nevertheless, it is possible that the availability of virtualized clusters will lead to renewed interest in the kernel-level SSI approach. Whether or not this comes to pass, it appears that the future of SSI is bound closely with that of virtualization and cloud technologies.

In terms of the virtualization *versus* SSI debate, the market has spoken – virtualization has been found to be useful as a mainstream general-purpose technology while kernel-level SSI has not. The isolation and consolidation provided by virtualization



(a) Number of publications containing the "single system image" trigram in the Google Books corpus (version 20090715) between 1980 and 2008. At the time of writing, n-gram frequency data is unavailable for years after 2008.



(b) Number of publications found for the "single system image" search term in Google Scholar for each publication year between 1980 and 2011. Data collected on September 27 2012.

Figure 2: Yearly frequency of publications containing the "single system image" trigram as a proxy for academic interest. A peak in frequency during the early-to-mid 2000s is evident.

clearly have major benefits, while the resource sharing and aggregation that characterize SSI would appear to be orthogonal to this approach. Much of the novel recent work on SSI has been on distributed hypervisors, which are themselves a form of virtualization and hence stand in contrast to the classic kernel-level approach. Increasingly, a trend is emerging towards a combination of outsourcing to public clouds and consolidation of in-house resources via virtualization. Barring a technological sea-change it is difficult to see how SSI operating systems will defy this trend to become widely installed on bare-metal hardware in the foreseeable future.

Despite the lack up uptake for kernel-level SSI, it must be remembered that there was a reason for the initial enthusiasm for the approach: it provides a very convenient method of parallelizing workloads that lend themselves to process migration or distributed shared memory. As noted above, it is clear that most organizations are not willing to devote resources to running a dedicated SSI cluster. However, there are some compelling advantages to a combination of both technologies, in the form of virtualized SSI clusters (see Section 4.6). This approach allows SSI clusters to be used only for those workloads at which they excel. The consolidation provided by virtualization allows clusters dedicated to other roles, such as message-passing and map/reduce processing, to exist simultaneously on the same physical hardware. SSI then becomes another tool in the parallel toolbox that is used where appropriate. Time will tell if the availability of easy-to-install virtual clusters will lead to renewed interest.

Acknowledgements

The authors wish to thank Ian Lee for helping to gather the data depicted in Figure 2. The research work described in this paper was supported by the Irish Centre for Cloud Computing and Commerce, an Irish national technology centre funded by Enterprise Ireland and the Irish Industrial Development Authority.

References

[1] R. Buyya, Single system image: need, approaches, and supporting HPC systems, in: The 1997 International Conference on Parallel and Distributed Processing Techniques and Applications, Proceedings, Las Vegas, Nevada, USA, 1997, pp. 1106–1113.

[2] R. Buyya, T. Cortes, H. Jin, Single system image, International Journal of High Performance Computing Applications 15 (2) (2001) 124–135.

[3] G. Pfister, The varieties of single system image, in: IEEE Workshop on Advances in Parallel and Distributed Systems, Proceedings, 1993, pp. 59–63.

[4] R. Lottiaux, B. Boissinot, P. Gallard, G. Vallée, C. Morin, OpenMosix, OpenSSI and Kerrighed: A comparative study, Research Report RR-5399, INRIA (2004).

[5] M. Baker, R. Buyya, Cluster computing at a glance, High Performance Cluster Computing: Architecture and Systems 1 (1999) 3–47.

[6] K. Hwang, G. C. Fox, J. J. Dongarra, Distributed and Cloud Computing: From Parallel Processing to the Internet of Things, Elsevier/Morgan Kaufmann, 2012.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, D. Yeung, The MIT Alewife machine: architecture and performance, in: 22nd Annual International Symposium on Computer Architecture, Proceedings, IEEE, 1995, pp. 2–13.

[8] D. Chaiken, J. Kubiatowicz, A. Agarwal, LimitLESS directories: A scalable cache coherence scheme, SIGOPS Operating Systems Review 25 (Special Issue) (1991) 224–234.

[9] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M. S. Lam, The Stanford Dash multiprocessor, Computer 25 (3) (1992) 63–79.

[10] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy, The Stanford FLASH multiprocessor, in: 21st Annual International Symposium on Computer Architecture, Proceedings, IEEE, 1994, pp. 302–313.

[11] B. C. Brock, G. D. Carpenter, E. Chiprout, M. E. Dean, P. L. De Backer, E. N. Elnozahy, H. Franke, M. E. Giampapa, D. Glasco, J. L. Peterson, R. Rajamony, R. Ravindran, F. L. Rawson III, R. L. Rockhold, J. Rubio, Experience with building a commodity Intel-based ccNUMA system, IBM Journal of Research and Development 45 (2) (2001) 207–227.

[12] Y. Koyanagi, T. Horie, T. Miyoshi, M. Ishii, Synfinity II - a high-speed interconnect with 2 GBytes/sec self-configurable physical link, in: Hot Interconnects 9., IEEE, 2001, pp. 23–29.

[13] A. Grbic, S. Brown, S. Caranci, R. Grindley, M. Gusat, G. Lemieux, K. Loveless, N. Manjikian, S. Srbljic, M. Stumm, Z. Vranesic, Z. Zilic, Design and implementation of the NUMAchine multiprocessor, in: 35th Annual Design Automation Conference, Proceedings, ACM, 1998, pp. 66–69.

[14] T. Lovett, R. Clapp, STiNG: A CC-NUMA computer system for the commercial marketplace, in: 23rd Annual International Symposium on Computer Architecture, Proceedings, IEEE, 1996, pp. 308–308.

[15] J. Laudon, D. Lenoski, The SGI Origin: a ccNUMA highly scalable server, in: ACM SIGARCH Computer Architecture News, Vol. 25, ACM, 1997, pp. 241–251.

[16] Unisys, Es7000/one enterprise server technical overview (Apr. 2007).

[17] M. Woodacre, D. Robb, D. Roe, K. Feind, The SGI® Altix™ 3000 global shared-memory architecture (2003).

[18] Hewlett-Packard, HP Integrity Superdome 2 user service guide (Sep. 2012).

[19] M. Côrrea, R. Chanin, A. Sales, R. Scheer, A. Zorzo, Multilevel load balancing in NUMA computers, in: 20th ACM Symposium on Operating Systems Principles, Proceedings, 2005, pp. 1–9.

[20] SGI, SGI® NUMAlink™industry leading interconnect technology, Tech. rep. (2005).

[21] ScaleMP, vSMP Foundation from ScaleMP, Datasheet (2014).

[22] D. Schmidl, C. Terboven, A. Wolf, D. A. Mey, C. Bischof, How to scale nested OpenMP applications on the ScaleMP vSMP architecture, in: IEEE International Conference on Cluster Computing (CLUSTER), Proceedings, IEEE, 2010, pp. 29–37.

[23] K. Kaneda, Y. Oyama, A. Yonezawa, A virtual machine monitor for providing a single system image, in: 17th IPSJ Computer System Symposium, Proceedings, 2005, pp. 3–12.

[24] M. Chapman, G. Heiser, Implementing transparent shared memory on clusters using virtual machines, in: USENIX Annual Technical Conference, Proceedings, USENIX Association, 2005, pp. 23–23.

[25] J. Peng, X. Long, L. Xiao, DVMM: A distributed VMM for supporting single system image on clusters, in: 9th International Conference for Young Computer Scientists, Proceedings, 2008, pp. 183 –188.

[26] Z. Song, L. Xiao, Research and design of inter-communication in DVMM, in: ISECS International Colloquium on Computing, Communication, Control, and Management, Proceedings, Vol. 4, IEEE, 2009, pp. 546–549.

[27] L. Yong, Single system image with virtualization technology for cluster computing environment, in: Third International Conference on Convergence and Hybrid Information Technology, Proceedings, Vol. 2, 2008, pp. 796–801.

[28] L. Ruan, J. Peng, L. Xiao, X. Wang, CloudDVMM: Distributed virtual machine monitor for cloud computing, in: Green Computing and Communications (GreenCom), IEEE, 2013, pp. 1853–1858.

[29] L. Ruan, J. Peng, L. Xiao, M. Zhu, Distributed virtual machine monitor for distributed cloud computing nodes integration, in: Grid and Pervasive Computing, Springer, 2013, pp. 23–31.

[30] X. Wang, M. Zhu, L. Xiao, Z. Liu, X. Zhang, X. Li, NEX: Virtual machine monitor level single system support in Xen, in: First International Workshop on Education Technology and Computer Science, Proceedings, Vol. 3, 2009, pp. 1047–1051.

[31] B. Ma, M. Zhu, L. Xiao, Implementation of single system image under virtualized environment, in: Scalable Computing and Communications; Eighth International Conference on Embedded Computing, Proceedings, 2009, pp. 232–237.

[32] S. Lankes, P. Reble, C. Clauss, O. Sinnen, The path to MetalSVM: Shared virtual memory for the scc, in: 4th Many-core Applications Research Community (MARC) Symposium, Proceedings, Potsdam, Germany, 2011.

[33] J.-A. Sobania, P. Tröger, A. Polze, Towards symmetric multi-processing support for operating systems on the scc, in: 4th Many-core Applications Research Community (MARC) Symposium, Proceedings, 2011.

[34] B. Walker, D. Steel, Implementing a full single system image UnixWare cluster: Middleware vs. underware, in: International Conference on Parallel and Distributed Processing Techniques and Applications, Proceedings, 1999.
[35] A. S. Tanenbaum, R. Van Renesse, Distributed operating systems, ACM Computing Surveys (CSUR) 17 (4) (1985) 419–470.
[36] B. Walker, G. Popek, R. English, C. Kline, G. Thiel, The LOCUS distributed operating system, ACM SIGOPS Operating Systems Review 17 (5) (1983) 49–70.
[37] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, et al., Overview of the Chorus distributed operating systems, in: Computing Systems, 1991.
[38] M. O'Connor, B. Tangney, V. Cahill, N. Harris, Micro-kernel support for migration, Distributed Systems Engineering 1 (4) (1994) 212.
[39] D. Cheriton, The V distributed system, Communications of the ACM 31 (3) (1988) 314–333.
[40] D. R. Cheriton, M. A. Malcolm, L. S. Melen, G. R. Sager, Thoth, a portable real-time operating system, Communications of the ACM 22 (2) (1979) 105–115.
[41] M. Nelson, B.-H. Lim, G. Hutchins, Fast transparent migration for virtual machines, in: USENIX Annual Technical Conference, Proceedings, 2005, pp. 391–394.
[42] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, B. Welch, The Sprite network operating system, Computer 21 (2) (1988) 23–36.
[43] S. J. Mullender, G. Van Rossum, A. Tananbaum, R. Van Renesse, H. Van Staveren, Amoeba: A distributed operating system for the 1990s, Computer 23 (5) (1990) 44–53.
[44] F. Douglis, J. K. Ousterhout, M. F. Kaashoek, A. S. Tanenbaum, A comparison of two distributed systems: Amoeba and Sprite, Computing Systems 4 (4) (1991) 353–384.
[45] A. Goscinski, M. Hobbs, J. Silcock, GENESIS: an efficient, transparent and easy to use cluster operating system, Parallel Computing 28 (4) (2002) 557–606.
[46] D. De Paoli, A. Goscinski, M. Hobbs, G. Wickham, The RHODOS microkernel, kernel servers and their cooperation, in: IEEE First International Conference on Algorithms and Architectures for Parallel Processing, Proceedings, Vol. 1, IEEE, 1995, pp. 345–354.
[47] E. Hendriks, BProc: The Beowulf distributed process space, in: 16th International Conference on Supercomputing, Proceedings, ACM, 2002, pp. 129–136.
[48] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, et al., An OSF/1 UNIX for massively parallel multicomputers, in: Winter USENIX Conference, Proceedings, 1993, pp. 449–468.
[49] Y. A. Khalidi, J. M. Bernabeu, V. Matena, K. Shirriff, M. Thadani, Solaris MC: a multicomputer OS, in: USENIX Annual Technical Conference, Proceedings, 1996, pp. 191–204.
[50] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, S. R. Radia, An overview of the Spring system, in: CompCon Spring'94, Proceedings, 1994, pp. 122–131.
[51] A. B. Barak, A. Shapir, UNIX with satellite processors, Software: Practice and Experience 10 (5) (1980) 383–392.
[52] A. Barak, A. Litman, MOS: a multicomputer distributed operating system, Software: Practice and Experience 15 (8) (1985) 725–737.
[53] A. Barak, NSMOS–MOS port to the Nationals 32000 family architecture, in: 2nd Israel Conference on Computer Systems and Software Engineering, Proceedings, 1987, pp. 1–8.
[54] A. Barak, O. Laden, Y. Yarom, The NOW MOSIX and its preemptive process migration scheme, Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments 7 (2) (1995) 5–11.
[55] A. Barak, O. La'adan, The MOSIX multicomputer operating system for high performance cluster computing, Future Generation Computer Systems 13 (4) (1998) 361–372.
[56] A. Barak, O. La'adan, A. Shiloh, Scalable cluster computing with MOSIX for Linux, 5th Annual Linux Expo, Proceedings.
[57] A. Barak, A. Shiloh, The MOSIX Cluster Operating System for High-Performance Computing on Linux Clusters, Multi-Clusters and Clouds (2012).
[58] L. Amar, A. Barak, Z. Drezner, M. Okun, Randomized gossip algorithms for maintaining a distributed bulletin board with guaranteed age properties, Concurrency and Computation: Practice and Experience 21 (15) (2009) 1907–1927.
[59] A. Keren, A. Barak, Opportunity cost algorithms for reduction of I/O and interprocess communication overhead in a computing cluster, IEEE Transactions on Parallel and Distributed Systems. 14 (1) (2003) 39–50.

[60] J. Sloan, High performance Linux clusters with OSCAR, Rocks, openMosix, and MPI, O'Reilly, 2004.
[61] J. Bilbao, G. Garate, A. Olozaga, A. del Portillo, Easy clustering with openMosix, in: 9th WSEAS International Conference on Computers, Proceedings, 2005, pp. 1–6.
[62] openMosix, openMosix project officially ends, Press Release, Tel Aviv (March 2008).
[63] B. S. Ahmed, K. Samsudin, A. R. Ramli, Architectural review of load balancing single system image, Journal of Computer Science 4 (9) (2008) 752.
[64] B. J. Walker, Open single system image (openSSI) Linux cluster project, Tech. rep. (2003).
[65] K. Thomas, Programming Locking Applications, IBM Corporation, 2001.
[66] B. Cahill, What is OpenGFS?, White Paper (2004).
[67] P. Schwan, Lustre: Building a file system for 1000-node clusters, in: 2003 Linux Symposium, Proceedings, 2003.
[68] G. Vallée, R. Lottiaux, L. Rilling, J.-Y. Berthou, I. D. Malhen, C. Morin, A case for single system image cluster operating systems: the Kerrighed approach, Parallel Processing Letters 13 (02) (2003) 95–122.
[69] C. Morin, P. Gallard, R. Lottiaux, G. Vallée, Towards an efficient single system image cluster operating system, Future Generation Computer Systems 20 (4) (2004) 505–521.
[70] P. Osiński, E. Niewiadomska-Szynkiewicz, Comparative study of single system image clusters, Evolutionary Computation and Global Optimization 169 (2009) 145–154.
[71] R. Lottiaux, C. Morin, Containers: A sound basis for a true single system image, in: First IEEE/ACM International Symposium on Cluster Computing and the Grid, Proceedings, IEEE, 2001, pp. 66–73.
[72] M. Novak, M. Fertré, J. Parpaillon, P. Linnell, Final prototype of LinuxSSI, Deliverable D2.2.11, XtreemOS (2010).
[73] M. Kačer, D. Langr, P. Tvrdík, Clondike: Linux cluster of non-dedicated workstations, in: IEEE International Symposium on Cluster Computing and the Grid, Proceedings, Vol. 1, IEEE, 2005, pp. 574–581.
[74] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, T. E. Anderson, GLUnix: A global layer Unix for a network of workstations, Software Practice and Experience 28 (9) (1998) 929–961.
[75] D. P. Ghormley, D. Petrou, S. H. Rodrigues, T. E. Anderson, SLIC: An extensibility system for commodity operating systems, in: USENIX Annual Technical Conference, Proceedings, Vol. 98, 1998.
[76] W. Yu, A. Cox, Java/DSM: A platform for heterogeneous computing, Concurrency: Practice and Experience 9 (11) (1997) 1213–1224.
[77] M. Lobosco, A. Silva, O. Loques, C. L. de Amorim, A new distributed JVM for cluster computing, in: Euro-Par 2003 – Parallel Processing, Springer, 2003, pp. 1207–1215.
[78] M. J. M. Ma, C.-L. Wang, F. C. M. Lau, JESSICA: Java-enabled single-system-image computing architecture, Journal of Parallel and Distributed Computing 60 (10) (2000) 1194–1222.
[79] M. Surdeanu, D. Moldovan, Design and performance analysis of a distributed Java virtual machine, IEEE Transactions on Parallel and Distributed Systems 13 (2002) 6.
[80] Y. Aridor, M. Factor, A. Teperman, cJVM: A single system image of a JVM on a cluster, in: 1999 International Conference on Parallel Processing, Proceedings, 1999, pp. 4–11.
[81] C. Tan, C. Tan, W. Wong, Shell over a cluster (SHOC): towards achieving single system image via the shell, in: 2002 IEEE International Conference on Cluster Computing, Proceedings, 2002, pp. 28 – 36.
[82] W. Zhang, Linux virtual server for scalable network services, in: Ottawa Linux Symposium, Proceedings, 2000, pp. 1–10.
[83] K. Shiraz, Red Hat Enterprise Linux Cluster Suite, Linux Journal 2007 (163).
[84] P. Krueger, M. Livny, A comparison of preemptive and non-preemptive load distributing, in: 8th International Conference on Distributed Computing Systems, Proceedings, 1988, pp. 123–130.
[85] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, S. Jiang, Current practice and a direction forward in checkpoint/restart implementations for fault tolerance, in: First Workshop on System Management Tools for Large-Scale Parallel Systems, Proceedings, Denver, CO, 2005.
[86] H. Zhong, J. Nieh, CRAK: Linux checkpoint/restart as a kernel module, Technical Report CUCS-014-01, Department of Computer Science, Columbia University (2001).
[87] J. Smith, A survey of process migration mechanisms, ACM SIGOPS Operating Systems Review 22 (3) (1988) 28–40.
[88] M. Nuttall, A brief survey of systems providing process or object migration facilities, ACM SIGOPS Operating Systems Review 28 (4) (1994) 64–80.

[89] D. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, Process migration, ACM Computing Surveys (CSUR) 32 (3) (2000) 241–299.
[90] M. Satyanarayanan, A survey of distributed file systems, Annual Review of Computer Science 4 (1) (1990) 73–104.
[91] E. Levy, A. Silberschatz, Distributed file systems: Concepts and examples, ACM Computing Surveys (CSUR) 22 (4) (1990) 321–374.
[92] T. D. Thanh, S. Mohan, E. Choi, S. Kim, P. Kim, A taxonomy and survey on distributed file systems, in: Fourth International Conference on Networked Computing and Advanced Information Management, Proceedings, Vol. 1, IEEE, 2008, pp. 144–149.
[93] L. Amar, A. Barak, A. Shiloh, The MOSIX direct file system access method for supporting scalable cluster file systems, Cluster Computing 7 (2) (2004) 141–150.
[94] K. Buytaert, et al., The OpenMosix HOWTO, The Linux Documentation Project.
[95] M. Šťava, P. Tvrdík, File system security in the environment of non-dedicated computer clusters, in: Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies, Proceedings, IEEE, 2007, pp. 445–452.
[96] R. S. C. Ho, K. Hwang, H. Jin, Single I/O space for scalable cluster computing, in: 1st IEEE Computer Society International Workshop on Cluster Computing, Proceedings, 1999, pp. 158–166.
[97] E. K. Lee, C. A. Thekkath, Petal: distributed virtual disks, SIGOPS Operating System Review 30 (5) (1996) 84–92.
[98] Z.-l. Jiang, M.-f. Zhu, L.-m. Xiao, An approach to implementing the NIC virtualization by the hybrids of single system image and hardware-assisted virtualization technologies, in: ISECS International Colloquium on Computing, Communication, Control, and Management, 2009, Proceedings, Vol. 4, 2009, pp. 577–582.
[99] A. Barak, A. Shiloh, The Virtual OpenCL (VCL) cluster platform, in: Intel European Research & Innovation Conference, Proceedings, Leixlip, Ireland, 2011, pp. 196–200.
[100] A. Barak, T. Ben-Nun, E. Levy, A. Shiloh, A package for OpenCL based heterogeneous computing on clusters with many GPU devices, in: IEEE International Conference on Cluster Computing, Proceedings, 2010, pp. 1–7.
[101] J. M. Gosney, Password cracking HPC, in: Passwords^12, Oslo, Norway, 2012.
[102] SGI, Reconfigurable Application-Specific Computing User's Guide (2008).
[103] H. Cofer, M. Fouquet-Lapar, T. Gamerdinger, C. Lindahl, B. Losure, A. Mayer, J. Swoboda, T. Utsumi, Creating the world's largest reconfigurable supercomputing system based on the scalable SGI® Altix® 4700 system infrastructure and benchmarking life-science applications, Reconfigurable Computing: Architectures, Tools and Applications (2008) 268–273.
[104] I. Latter, Security and openMosix; securely deploying SSI cluster technology over untrusted networking infrastructure, White Paper, Macquarie University (2003).
[105] M. Kačer, P. Tvrdík, Protecting non-dedicated cluster environments by marking processes with stigmata, in: International Conference on Advanced Computing and Communications, 2006, Proceedings, IEEE, 2006, pp. 107–112.
[106] A. Barak, A. Shiloh, L. Amar, An organizational grid of federated MOSIX clusters, in: 2005 IEEE International Symposium on Cluster Computing and the Grid, Proceedings, Vol. 1, 2005, pp. 350–357.
[107] L. Amar, J. Stößer, E. Levy, A. Shiloh, A. Barak, D. Neumann, Harnessing migrations in a market-based grid OS, in: 9th IEEE/ACM International Conference on Grid Computing, Proceedings, 2008, pp. 85–94.
[108] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, A. Reinefeld, XtreemOS: A vision for a grid operating system, Technical Report #4, XtreemOS (2008).
[109] J. Mehnert-Spahn, T. Ropars, M. Schoettner, C. Morin, The architecture of the XtreemOS grid checkpointing service, Euro-Par 2009 – Parallel Processing (2009) 429–441.
[110] R. J. Creasy, The origin of the VM/370 time-sharing system, IBM Journal of Research and Development 25 (5) (1981) 483–490.
[111] R. P. Goldberg, Architecture of virtual machines, in: Workshop on Virtual Computer Systems, Proceedings, ACM, 1973, pp. 74–112.
[112] J. S. Robin, C. E. Irvine, Analysis of the Intel Pentium's ability to support a secure virtual machine monitor, in: 9th conference on USENIX Security Symposium, Proceedings, 2000.
[113] A. Liguori, The myth of Type I and Type II hypervisors, Tales of a Code Monkey (Blog), Online; accessed July 24 2013.

[114] S. Soltesz, H. Pötzl, M. Fiuczynski, A. Bavier, L. Peterson, Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors, ACM SIGOPS Operating Systems Review 41 (3) (2007) 275–287.
[115] J. Gallard, G. Vallée, A. Lèbre, C. Morin, P. Gallard, S. L. Scott, Complementarity between virtualization and single system image technologies, in: Euro-Par 2008 Workshops – Parallel Processing, Springer, 2009, pp. 43–52.
[116] J. Gallard, A. Lèbre, G. Vallée, P. Gallard, L. Scott, Stephen, C. Morin, Is virtualization killing single system image research?, Research Report RR-6389, INRIA (2007).
[117] T. Maoz, A. Barak, L. Amar, Combining virtual machine migration with process migration for HPC on multi-clusters and grids, in: 2008 IEEE International Conference on Cluster Computing, Proceedings, 2008, pp. 89–98.
[118] L. M. Vaquero, L. Rodero-Merino, R. Buyya, Dynamically scaling applications in the cloud, ACM SIGCOMM Computer Communication Review 41 (1) (2011) 45–52.
[119] P. Marshall, H. Tufo, K. Keahey, D. LaBissoniere, M. Woitaszek, Architecting a large-scale elastic environment: Recontextualization and adaptive cloud services for scientific computing, in: 7th International Conference on Software Paradigm Trends, Proceedings, Rome, Italy, 2012.
[120] C. Morin, Y. Jégou, J. Gallard, P. Riteau, Clouds: a new playground for the XtreemOS grid operating system, Parallel Processing Letters 19 (03) (2009) 435–449.
[121] T. Kielmann, G. Pierre, C. Morin, XtreemOS: a sound foundation for cloud infrastructure and federations, Grids, P2P and Services Computing (2010) 1–5.
[122] P. Healy, J. Morrison, R. Walshe, ElasticSSI: Self-optimizing metacomputing through process migration and elastic scaling, ERCIM News 90.
[123] G. Pfister, Multi-multicore Single System Image/Cloud Computing. A good idea?, The Perils of Parallel (Blog), Online; accessed September 26 2012.
[124] D. Wentzlaff, C. Gruenwald III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, A. Agarwal, An operating system for multicore and clouds: mechanisms and implementation, in: ACM symposium on Cloud computing, Proceedings, 2010, pp. 3–14.
[125] D. Becker, B. Monkman, Scyld ClusterWare^{TM}: An innovative architecture for maximixing return on investment in Linux clustering, Penguin Computing (2006).
[126] R. Goeckelmann, M. Schoettner, S. Frenz, P. Schulthess, Plurix, a distributed operating system extending the single system image concept, in: Canadian Conference on Electrical and Computer Engineering, Proceedings, Vol. 4, 2004, pp. 1985–1988.
[127] G. Watson, M. Sottile, R. Minnich, S. Choi, E. Hertdriks, Pink: A 1024-node single-system image Linux cluster, in: Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region, Proceedings, 2004, pp. 454–461.
[128] N. Kofahi, S. Al Zahrani, S. M. Hussain, MOSIX evaluation on a Linux cluster, International Arab Journal of Information Technology 3 (1) (2006) 62–68.
[129] D. Margery, G. Vallée, R. Lottiaux, C. Morin, J.-Y. Berthou, Kerrighed: A SSI cluster OS running OpenMP, Research Report RR-4947, INRIA (2003).
[130] G. Krawezik, F. Cappello, Performance comparison of MPI and three openMP programming styles on shared memory multiprocessors, in: Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, Proceedings, 2003, pp. 118–127.
[131] G. Pfister, In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing, Prentice-Hall, 1998.
[132] K. Hwang, Z. Xu, Scalable Parallel Computing: Technology, architecture, programming, McGraw-Hill, 1998.
[133] B. Urgaonkar, A. Rosenberg, P. Shenoy, Application placement on a cluster of servers, International Journal of Foundations of Computer Science 18 (05) (2007) 1023–1041.
[134] C. Tang, M. Steinder, M. Spreitzer, G. Pacifici, A scalable application placement controller for enterprise data centers, in: 16th International Conference on World Wide Web, Proceedings, 2007, pp. 331–340.
[135] A. Rotem-Gal-Oz, Fallacies of distributed computing explained, White Paper (2006).
[136] J. Spolsky, Joel on Software, Apress, 2004.
[137] A. Muller, S. Wilson, Virtualization with VMware ESX Server, Syngress Publishing, 2005.
[138] E. M. Rogers, The Diffusion of Innovations, Fifth Edition, Simon and Schuster, 2003.
[139] R. Depietro, E. Wiarda, M. Fleischer, The context for change: Organization, technology and environment, The Processes of Technological Innovation (1990) 151–175.
[140] C. L. Iacovou, I. Benbasat, A. S. Dexter, Electronic data interchange and small organizations: adoption and impact of technology, MIS Quarterly (1995) 465–485.

[141] S. Davies, The Diffusion of Process Innovations, CUP Archive, 1979.
[142] P. Attewell, Technology diffusion and organizational learning: The case of business computing, Organization Science 3 (1) (1992) 1–19.
[143] E. von Hippel, The Sources of Innovation, Oxford University Press, 1988.
[144] A. Davila, M. Gupta, R. Palmer, Moving procurement systems to the Internet: the adoption and use of e-procurement technology models, European Management Journal 21 (1) (2003) 11–23.
[145] J. D. Eveland, L. G. Tornatzky, The deployment of technology, The Processes of Technological Innovation (1990) 117–148.
[146] F. D. Davis, R. P. Bagozzi, P. R. Warshaw, Extrinsic and intrinsic motivation to use computers in the workplace, Journal of Applied Social Psychology 22 (14) (1992) 1111–1132.
[147] M. Obal, Why do incumbents sometimes succeed? Investigating the role of interorganizational trust on the adoption of disruptive technology, Industrial Marketing Management 42 (6) (2013) 900–908.
[148] P. Klemperer, Markets with consumer switching costs, The Quarterly Journal of Economics 102 (2) (1987) 375–394.
[149] M. L. Katz, C. Shapiro, Network externalities, competition, and compatibility, The American Economic Review 75 (3) (1985) 424–440.
[150] R. G. Fichman, C. F. Kemerer, Adoption of software engineering process innovations: The case of object-orientation, Sloan Management Review 34 (2).
[151] T. Oliveira, M. F. Martins, Literature review of information technology adoption models at firm level, The Electronic Journal Information Systems Evaluation 14 (1) (2011) 110–121.
[152] J. Dedrick, J. West, Why firms adopt open source platforms: a grounded theory of innovation and standards adoption, in: MISQ Special Issue Workshop: Standard Making: A Critical Research Frontier for Information Systems, Seattle, WA, 2003, pp. 236–257.
[153] J. L. Bower, C. M. Christensen, Disruptive technologies: catching the wave, Harvard Business Review, 1995.
[154] C. Christensen, The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail, Harvard Business Press, 1997.
[155] D. K. Rigby, C. M. Christensen, M. Johnson, Foundations for growth: How to identify and build disruptive new businesses, MIT Sloan Management Review 43 (3) (2002) 22–32.
[156] J. Michel, Y. Shen, A. Aiden, A. Veres, M. Gray, J. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, et al., Quantitative analysis of culture using millions of digitized books, Science 331 (6014) (2011) 176–182.
[157] J. Farrell, G. Saloner, Standardization, compatibility, and innovation, The RAND Journal of Economics (1985) 70–83.