

Scalability vs. Development Effort for Multiprocessor Operating System Kernels

Simon Kågström, Håkan Grahn, and Lars Lundberg

Department of Systems and Software Engineering, School of Engineering
Blekinge Institute of Technology, Ronneby, Sweden

{ska, hgr, llu}@bth.se

Abstract

With multiprocessors becoming increasingly common, many operating systems have to be adapted to work with the multiprocessor systems. In this paper, we present a categorization of porting methods for multiprocessor operating systems. We also perform a case study of the evolution of multiprocessor support for the Linux kernel, both in terms of performance and implementation complexity.

1 Introduction

Shared memory multiprocessors are becoming very common, making operating system support for multiprocessors increasingly important. Some operating systems already include support for multiprocessors, but many special-purpose operating systems still need to be ported to benefit from multiprocessor computers.

There are a number of possible technical approaches when porting an operating system to a multiprocessor, e.g., introducing coarse or fine grained locking of shared resources in the kernel, introducing a virtualization layer between the operating system and the hardware, using master-

slave or other asymmetric solutions.

The development effort and lead time for porting an operating system to a multiprocessor varies much depending on the technical approach used. The technical approach also affects multiprocessor performance. The choice of technical approach depends on a number of factors; two important factors are the available resources for doing the port and the performance requirements on the multiprocessor operating system. Consequently, understanding the performance and development cost implications of different technical solutions is crucial when selecting a suitable approach.

In this paper we identify seven technical approaches for doing a multiprocessor port. We also provide an overview of the development time and multiprocessor performance implications of each of these approaches. We base our results on a substantial literature survey and a case study concerning the development effort and multiprocessor performance (in terms of scalability) of different versions of Linux. We have limited the study to operating systems for shared memory multiprocessors.

The rest of the paper is structured as follows. Section 2 describes the challenges faced in a multiprocessor port. In Section 3 we present a categorization of porting methods, and Section 4 then categorizes a number of existing multiprocessor systems. In Section 5, we describe our Linux case study, and finally discuss our findings and conclude in Section 6.

2 Multiprocessor Port Challenges

A uniprocessor operating system need to be changed in a number of ways to support multiprocessor hardware. Some data structures must be made CPU-local, e.g., the kernel stack, the currently running thread. The way this is implemented varies. One method is to replace the affected variables with vectors. Another is to cluster the CPU-local data structures in a single virtual page and map this page to different physical memory pages for every CPU.

In a non-preemptible uniprocessor kernel, the only source of concurrency issues is interrupts and interrupt masking is then enough for protection against concurrent access. On multiprocessors, disabling interrupts is not enough as it only affects the local processor and locking is needed instead. Kernels which support in-kernel process preemption need protection of shared data structures even

on uniprocessors.

Finally, since many modern processors employ memory access reordering to improve performance, memory barriers are sometimes needed to prevent inconsistencies. For example, a structure need to be written into memory before the structure is inserted into a list for all processors to see the updated data structure.

3 Categorization

We have categorized the porting approaches into the following: *giant locking*, *coarse-grained locking*, *fine-grained locking*, *lock-free*, *asymmetric*, *virtualization*, and *reimplementation*. Other surveys and books [21, 24] use other categorizations, e.g., depending on the structuring approach (micro-kernels and monolithic kernels). Figure 1 illustrates the different methods as well as a few specific implementations (described in Section 4).

3.1 Giant Locking

With giant locking (Figure 1a), a single spin lock protects the entire kernel from concurrent access. The giant lock serializes all kernel accesses, so most of the uniprocessor semantics can be kept. Giant locking requires small changes to the kernel apart from acquiring and releasing the lock, e.g., processor-local pointers to the current process. Performance-wise, scalability is limited by having only one processor executing in the kernel at a time.

3.2 Coarse-grained Locking

Coarse-grained locks protect larger collections of data or code, such as entire subsystems as shown in Figure 1b. Compared to giant-locking, coarse-grained locks open up for some parallel work in the kernel. For example, a coarse-grained kernel can have separate locks for the filesystem and network subsystems, allowing two processors to execute in different subsystems. However, inter-dependencies between the subsystems can force an effective serialization similar to giant locking. If

subsystems are reasonably self-contained, coarse-grained locking is fairly straightforward, otherwise complex dependencies might cause data races or deadlocks.

A special case of coarse-grained locking is *funnels*, used in DEC OSF/1 [6], and Mac OS X [8]. In OSF/1, code running inside a funnel always executes serialized on a “master” processor, similar to master-slave systems. Mac OS X does not restrict the funnel to a single processor. Instead the funnel acts as a subsystem lock, which is released on thread rescheduling.

3.3 Fine-grained Locking

Fine-grained locking (Figure 1c), restricts the locking to individual data structures or even parts of data structures. Fine-grained locking allows for increased parallelism at the cost of more lock invocations and more complex engineering. Even fine-grained implementations will sometimes use coarse-grained locks, which are more beneficial for uncontended data.

3.4 Lock-free Approaches

Using hardware support, it is possible to construct lock-free operating systems. Lock-free algorithms rely on instructions for atomically checking and updating a word in memory (compare-and-swap, CAS), found on many CPU architectures. However, for efficient implementation of lock-free algorithms, a CAS instruction capable of updating multiple locations is needed, e.g., double CAS (DCAS). Simple structures such as stacks and lists can be implemented directly with CAS and DCAS, while more complex structures use versioning and retries to detect and handle concurrent access.

A completely lock-free operating system relies on these specialized data structures. Lock-free data structures are sometimes hard to get correct and can be inefficient without proper hardware support [7], which limits the scalability of completely lock-free approaches. Also, making an existing kernel lock-free requires a major refactoring of the kernel internals, so the development costs of a lock-free kernel is likely to be high.

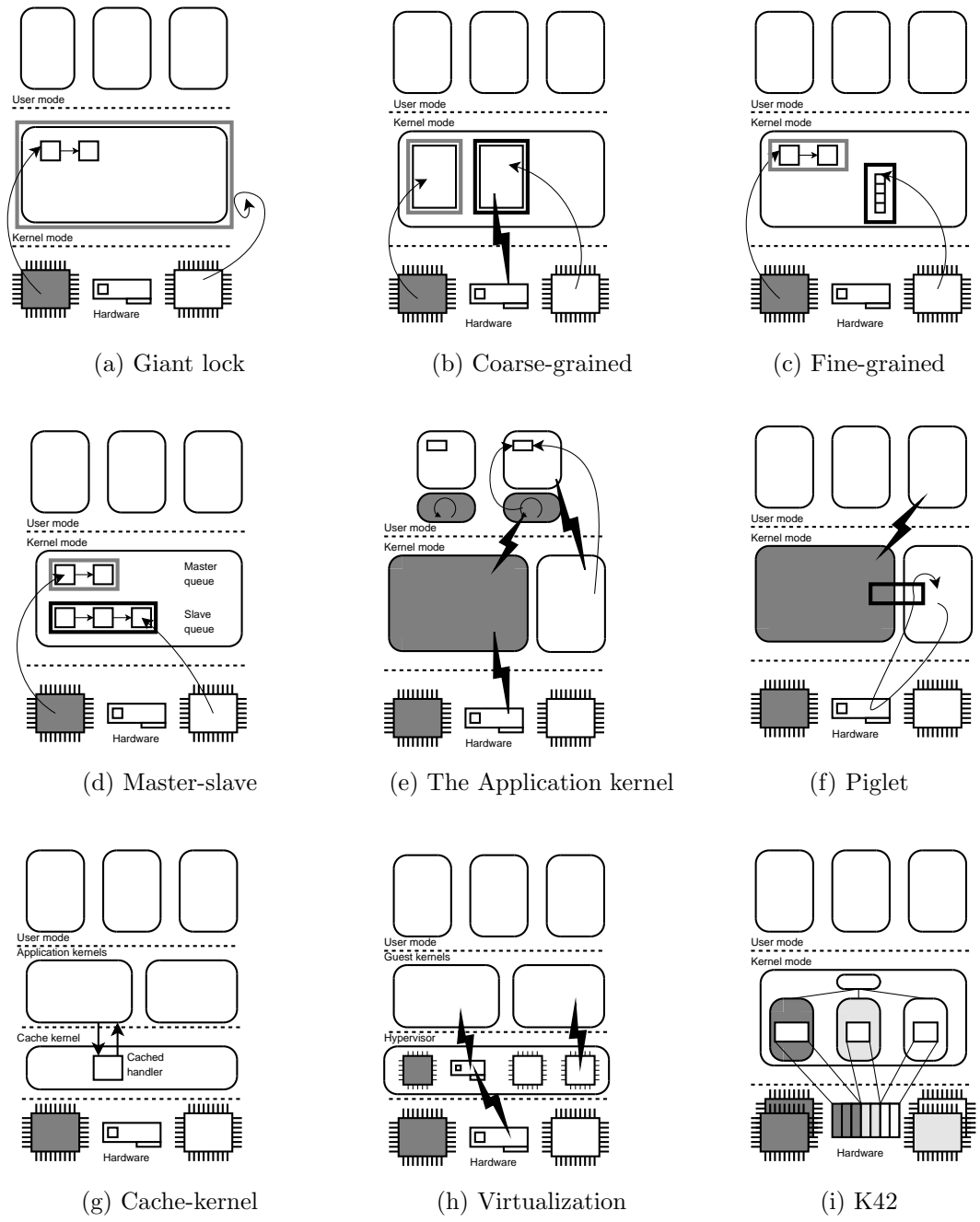


Figure 1: Multiprocessor operating system organizations. Thick lines show locks and the flash symbol denote system calls or device interrupts. The figure shows both categories and examples of systems.

3.5 Asymmetric Approaches

It is also possible to divide the work asymmetrically among the processors. Asymmetric operating systems assign processors to special uses, e.g., compute processors or I/O handling processors. Since asymmetric systems can be very diverse, both the implementation cost and scalability of these systems will vary.

3.6 Virtualization

A completely different method is to partition the multiprocessor machine into a virtual cluster, running many OS instances on shared hardware (Figure 1h). This category can be further subdivided into fully virtualized and paravirtualized systems, where the latter employs operating system modifications and virtual extensions to the architecture to handle hardware deficiencies or lower virtualization overhead.

The virtualizing layer is called a Hypervisor. The Hypervisor performs handling and multiplexing of virtualized resources, which makes the Hypervisor fairly straightforward to implement. As virtualization also allows existing uniprocessor operating systems to run with small or no modifications, the development costs of a port is limited.

3.7 Reimplementation

A final approach is to reimplement the core of the kernel for multiprocessor support and provide API/ABI compatibility with the original kernel. While drastic, this can be an alternative for moving to large-scale multiprocessors, when legacy code might otherwise limit the scalability.

4 Operating System Implementations

In this section, we discuss different implementations of multiprocessor ports. Table 1 provides a summary of the discussed systems.

System	Method	Focus	Performance		Effort	
			Latency	Scalability	Code lines	Development time
Linux 2.0 [3]	Giant	General purpose	High	Low	955K	12 months
FreeBSD 4.9 [15]	Giant	General purpose	High	Low	1.9M	?
QNX [22]	Giant	Real-time	Low	?	?	?
Linux 2.2	Coarse	General purpose	Medium	Low	2.5M	18 months
Mac OS X [8]	Coarse	General purpose	High	Low	?	?
OSF/1 [6]	Fine	General purpose	Low	High	?	?
Linux 2.4	Fine	General purpose	Medium	Medium	5.2M	11 months
Linux 2.6 [16]	Fine	General purpose	Low	High	6.1M	11 months
AIX [5, 25]	Fine	General purpose	Low	High	?	18 months
Solaris [12]	Fine	General purpose	Low	High	?	?
FreeBSD 5.4	Fine	General purpose	Low	High	2.4M	?
Synthesis [18]	Lock-free	General purpose	Low	?	?	?
Cache kernel [4]	Lock-free	Application specific	Low	?	15K	?
Dual VAX 11/780 [9]	Asymmetric	General purpose	High	Low	?	?
Application kernel [14]	Asymmetric	Low effort	High	Low	3,600	5 weeks
Piglet [20]	Asymmetric	I/O intensive	As UP	Depends on UP	?	?
Cellular Disco [10]	Virtualized	Hardware sharing, fault tolerance	As Guest	As Guest	50K	?
VMWare ESX [23]	Virtualized	Hardware sharing	As Guest	As Guest	?	?
L4Linux [26]	Virtualized	Hardware sharing	As Guest	As Guest	?	?
Adeos [27]	Virtualized	Hardware sharing	As Guest	As Guest	?	?
Xen 2.05 [2]	Virtualized	Hardware sharing	As Guest	As Guest	75K+38K	?
K42 [1]	Reimplementation	Scalability	Low	High	50K	?

Table 1: Summary of the categorized multiprocessor operating systems. The code lines refer to the latest version available and the development time is the time between the two last major releases.

4.1 Locking-based Implementations

Many to multiprocessor ports of uniprocessor operating systems are first implemented with a giant lock approach, e.g., Linux 2.0 [3], FreeBSD [15], and other kernels [13], and later relaxes the locking scheme with a more fine-grained approach. The QNX Neutrino microkernel [22], also protect the kernel with a giant lock. However, as most operating system functionality is handled by server processes, the parallelization of these is more important than the actual kernel.

Mac OS X started out with what was effectively a giant lock (a funnel for the entire BSD portion of the kernel), but thereafter evolved into a more coarse-grained implementation with separate funnels for the filesystem and network subsystems. Currently, Mac OS X is reworked to support locking at a finer granularity.

AIX [5] and DEC OSF/1 3.0 [6] were released with fine-grained locking from the start. In both cases, the SMP port was based on a preemptible uniprocessor kernel, which simplified porting since disabling of preemption correspond to places where a lock is needed in the multiprocessor version. During the development of OSF/1, funneling was used to protect the different subsystems while core parts like the scheduler and virtual memory were parallelized. Solaris [12] and current versions of Linux [16] and FreeBSD also implement fine-grained locking.

4.2 Lock-free Implementations

To our knowledge, there exists only two operating system kernels which rely solely on lock-free algorithms; Synthesis [18] and the Cache Kernel [4]. Synthesis uses a traditional monolithic structure but restricts kernel data structures to a few simple lock-free implementations of e.g., queues and lists.

The Cache Kernel [4] (Figure 1g), was also implemented lock-free. The Cache Kernel provides basic kernel support for address spaces, threads, and application kernels. Instead of providing full implementations of these concepts, the Cache Kernel caches a set of active objects which are installed by the application kernels. For example, the currently running threads are present as thread objects holding the basic register state, while an application kernel holds the complete state. The Cache

Kernel design caters for a very small kernel which can be easily verified and implemented in a lock-free manner. Note, however, that the application kernels still need to be parallelized to fully benefit from multiprocessor operation. Both Synthesis and the Cache Kernel were implemented for the Motorola 68k CISC architecture, which has architectural support for DCAS. Implementations for other architectures which lack DCAS support might be more difficult.

4.3 Asymmetric Implementations

The most common asymmetric systems have been master-slave systems [9], which employ one master processor to run kernel code while the other (“slave”) processors only execute user space applications. The changes to the original OS in a master-slave port are mainly the introduction of separate queues for master and slave jobs, as shown in Figure 1d. Like giant locks, the performance of master-slave systems is limited by allowing only one processor in the kernel.

The Application kernel approach [14] (Figure 1e) allows keeping the original uniprocessor kernel as-is. The approach runs the original unmodified kernel on one processor, while user-level applications run on a small custom kernel on the other processors. All processes are divided in two parts, application threads and one bootstrap thread. The application threads run the original application, while the bootstrap thread forwards system calls, page faults etc., to the uniprocessor kernel.

Piglet [20] (Figure 1f) dedicates the processors to specific operating system functionality. Piglet allocates processors to run a Lightweight Device Kernel (LDK), which normally handles access to hardware devices but can perform other tasks. The LDK is not interrupt-driven, but instead polls devices and message buffers for incoming work. A prototype of Piglet has been implemented to run beside Linux 2.0.30, where the network subsystem (including device handling) has been offloaded to the LDK, and the Linux kernel and user-space processes communicate through lock-free message buffers with the LDK.

4.4 Virtualization

There are a number of virtualization systems. VMWare ESX server [23] is a fully virtualized system which uses execution-time dynamic binary translation to handle the deficiencies of the IA-

32 platform. Cellular disco [10] is a paravirtualized system created to use large NUMA machines efficiently. The underlying Hypervisor is divided into isolated cells, each handling a subset of the hardware resources to provide fault containment.

Xen [2] uses a paravirtualized approach for the Intel IA-32 architecture currently capable of running uniprocessor Linux and NetBSD as guest OS:es. The paravirtualized approach allows higher performance on IA-32. For example, the real hardware MMU can be used instead of software lookup in a virtual MMU. The Xen hypervisor implementation consists of around 75,000 lines of code while the modifications to Linux 2.6.10, mostly being the addition of a virtual architecture for Xen, is around 38,000 lines of code. The Adeos Nanokernel [27] also works similar to Xen, requiring modifications to the guest kernel's (Linux) source code.

4.5 Reimplementation

As an example of a reimplementation, K42 [1] (Figure 1i) is ABI-compatible with Linux but implemented from scratch. K42 is a microkernel-based operating system with most of the operating system functionality executing in user-mode servers or replaceable libraries. K42 avoids global objects and instead distributes objects among processors and directs access to the processor-local objects. Also, K42 supports runtime replacement of object implementations to improve performance for various workloads.

5 Linux Case Study

Linux evolved from using a giant locking approach in the 2.0 version, through a coarse-grained approach in 2.2 to using a more fine-grained approach in 2.4 and 2.6. Multiprocessor support in Linux was introduced in the stable 2.0.1 kernel, released in June 1996. 18 months later, in late January 1999, 2.2.0 was released. The 2.4.0 kernel came 11 months later, in early January 2001, while 2.6.0 was released in late December 2003, almost 12 months after the previous release.

We have studied how three parameters have evolved from kernel versions 2.0 to 2.6. First, we examined the locking characteristics. Second, we examined the source code changes for multiprocessor

Table 2: Number of locks in the Linux kernel.

Version	Number of locks					
	BKL	spinlock	rwlock	seqlock	rcu	sema
2.0.40	17	0	0	0	0	49
2.2.26	226	329	121	0	0	121
2.4.30	193	989	300	0	0	332
2.6.11.7	101	1,717	349	56	14	650

support, and third, we measured performance in a kernel-bound benchmark.

We chose to compare the latest versions of each of the stable kernel series, 2.0.40, 2.2.26, 2.4.30, and 2.6.11.7. We examined files in `kernel/`, `mm/`, `arch/i386/`, `include/asm-i386/`, i.e., the kernel core and the IA-32-specific parts. We also include `fs/` and `fs/ext2`. We chose the ext2 filesystem since it is available in all compared kernel versions. We exclude files implementing locks, e.g, `spinlocks.c`, and generated files.

To see how SMP support changes the source code, we ran the C preprocessor on the kernel source, with and without `_SMP_` and `CONFIG_SMP` defined. The preprocessor ran on the file only (without include-files). We also removed empty lines and indented the files with the `indent` tool to avoid changes in style.

5.1 Evolution of Locking in Linux

In the 2.0 versions, Linux uses a giant lock, the “Big Kernel Lock” (BKL). Interrupts are also routed to a single CPU, which further limits the performance. On the other hand, multiprocessor support in Linux 2.0 was possible to implemented without major restructuring of the uniprocessor kernel.

Linux 2.2 relaxed the giant locking scheme to adopt a coarse-grained scheme. 2.2 also added general-purpose basic spinlocks and spinlocks for multiple-readers / single-writer (rwlocks). The 2.2 kernels has subsystem locks e.g., for block device I/O requests, while parts of the kernel are protected at a finer granularity, e.g., filesystem inode lists and the runqueue. However, the 2.2 kernels still use the BKL for many operations, e.g., file read and write.

The 2.4 version of the kernel further relaxes the locking scheme. For example, the BKL is no

longer held for virtual file system reads and writes. Like earlier versions, 2.4 employs a single shared runqueue from which all processors take jobs.

Many improvements of the multiprocessor support were added in the 2.6 release. 2.6 introduced seqlocks, read-copy update mutual exclusion [19], processor-local runqueues, and kernel preemption. Kernel preemption allows processes to be preempted within the kernel, which reduces latency. A seqlock is a variant of rwlocks that prioritizes writers over readers. Read-copy update, finally, was carried over from K42 and is used to defer updates to a structure until a safe state when all active references to that structure are removed, which allows for lock-free access. The safe state is when the process does a voluntary context switch or when the idle loop is run, after which the updates can proceed.

5.2 Locking and Source Code Changes

Table 2 shows the how the lock usage has evolved throughout the Linux development. The table shows the number of places in the kernel where locks are acquired and released. Semaphores (sema in the table) are often used to synchronize with user-space, e.g., in the system call handling, and thus have the same use on uniprocessors.

As the giant lock in 2.0 protects the entire kernel, there are only 17 places with BKL operations (on system calls, interrupts, and in kernel daemons). The coarse-grained approach in 2.2 is significantly more complex. Although 2.2 introduced a number of separate spinlocks, around 30% (over 250 places) of the lock operations still handle the giant lock. The use of the giant lock has been significantly reduced in 2.4, with around 13% of the lock operations handling the giant lock. This trend continues into 2.6, where less than 5% of the lock operations handled the giant lock. The 2.6 seqlocks and read-copy update mutual exclusion are still only used in a few places in the kernel.

Table 3 shows the results from the C preprocessor study. From the table, we can see that most files do not contain explicit changes for the multiprocessor support. In terms of modified, added, or removed lines, multiprocessor support for the 2.0 kernel is significantly less intrusive than the newer kernels, with only 541 source lines (1.19% of the uniprocessor source code) modified. In 2.2 and 2.4, around 2.2% of the lines differ between the uniprocessor and multiprocessor kernels, while

Table 3: Lines of code with and without SMP support in Linux.

Version	Files	Changed Files	Lines		Modified/ new/ removed
			No SMP	SMP	
2.0.40	173	22	45,392	45,770	541
2.2.26	226	36	52,294	53,281	1,156
2.4.30	280	38	64,293	65,552	1,374
2.6.11.7	548	49	104,147	105,846	1,812

the implementation is closer again in 2.6 with 1.7% of the lines changed.

5.3 Performance Evaluation

We also did a performance evaluation to compare the different Linux kernel versions in the Postmark benchmark [11]. The Postmark benchmark models the file system behavior of Internet servers for electronic mail and web-based commerce, focusing on small-file performance. This kernel-bound benchmark requires a highly parallelized kernel to exhibit performance improvements (especially for the file and block I/O subsystems). We ran Postmark with 10,000 transactions, 1,000 simultaneous files and a file size of 100 bytes.

We compiled the 2.0 and 2.2 kernels with GCC 2.7.2 whereas 2.4 and 2.6 were compiled with GCC 3.3.5. All kernels were compiled with SMP-support enabled, which is a slight disadvantage on uniprocessors. The system is a minimal Debian GNU/Linux system which uses the ext2 filesystem. We ran 8 Postmark processes in parallel and measured the time used for all of them to complete. The benchmark was executed in the Simics full-system simulator [17], which was configured to simulate between 1 and 8 processors. Simics simulates a complete computer system including disks, network, and CPUs including the memory hierarchy modeled after the Pentium 4.

Figure 2 presents the scalability results for the Postmark benchmark normalized to uniprocessor performance in Linux 2.0. First, we can see that the absolute uniprocessor performance has increased, with 2.4 having the best performance. Linux 2.0 and 2.2 does not scale at all with this benchmark, while 2.4 shows some improvement over uniprocessor mode. On the other hand, 2.6

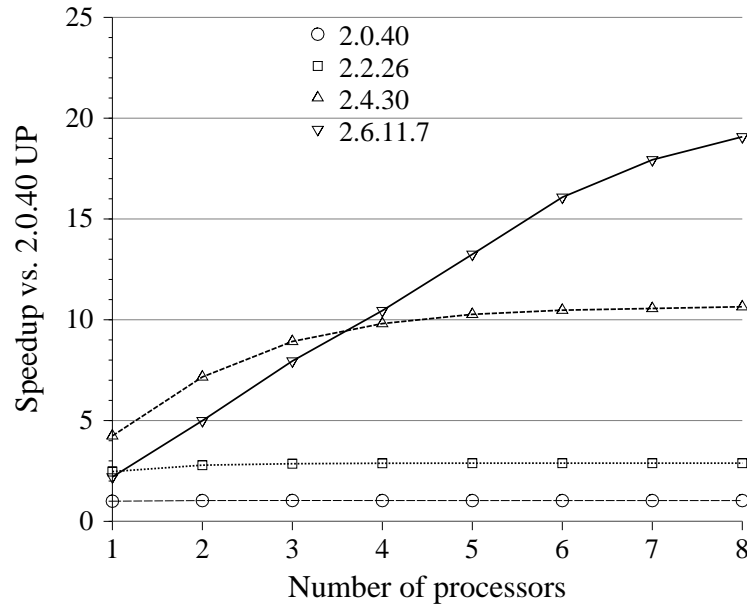


Figure 2: Postmark benchmark running on different versions of Linux.

scales well all the way to 8 processors. Since the 2.0 kernels use the giant locking approach, it is not surprising that it does not scale in this kernel-bound benchmark. Since much of the file subsystem in 2.2 still uses the giant lock and no scalability improvement is shown. The file subsystem revision in 2.4 gives it a slight scalability advantage, although it does not scale beyond 3 processors. It is not until the 2.6 kernel that Linux manages to scale well for the Postmark benchmark. Linux 2.6 has a very good scalability, practically linear until 7 processors.

6 Discussion and Conclusions

Figure 3 shows an approximation of the trade-off between scalability and effort of the categorizes presented. It should be noted that when porting an uniprocessor kernel to a multiprocessor, it is not always possible to freely select the porting approach. For example, employing the Xen hypervisor is only possible if the uniprocessor kernel is written for one of the architectures which Xen supports

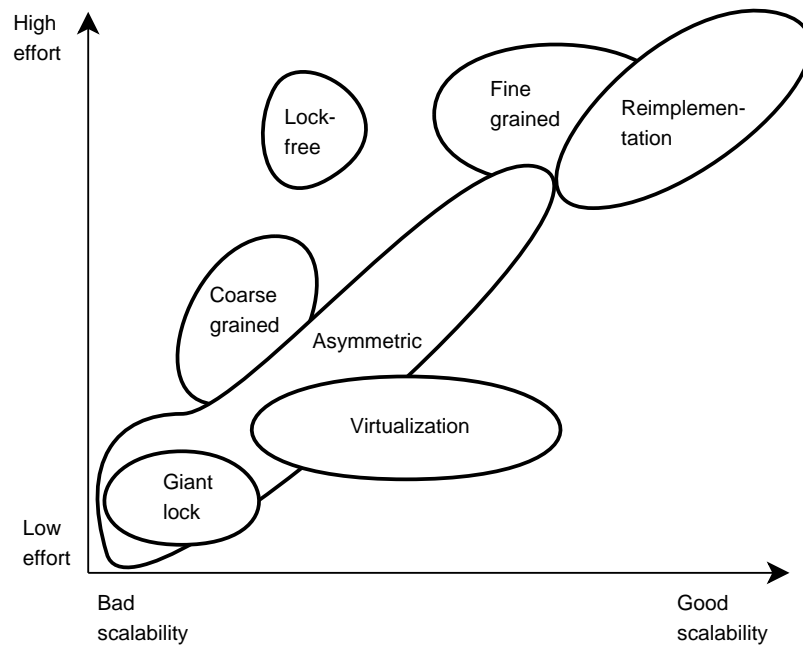


Figure 3: The available design space.

(currently IA-32).

The giant locking approach provides a very straightforward way of adding multiprocessor support since most of the uniprocessor semantics of the kernel can be kept. However, the kernel also becomes a serialization point, which makes scaling very difficult for kernel-bound benchmarks. As a foundation for further improvements, giant locking still provides a viable first step because of its relative simplicity.

Coarse-grained locking is more complex to introduce than giant locking, as seen in the Linux case study. Fine-grained locking further adds to the complexity, but also enables better scalability. The AIX and OSF/1 experiences indicates that a preemptible uniprocessor kernel simplifies multiprocessor porting with finer granularity locks.

Since asymmetric systems are very diverse, both scalability and effort will vary depending on the approach. In one extreme, the application kernel provides a generic porting method with low effort at the cost of bad scalability for kernel-bound applications. Master-slave systems require

more modifications to the original kernel, but have slightly better performance than the application kernel. More complex asymmetric systems, such as Piglet, can have good scalability on I/O-intensive workloads.

Because of complex algorithms and limited hardware support, completely lock-free operating systems require high effort to provide good scalability for ports of existing uniprocessor systems. Lock-free algorithms are still used in many lock-based operating systems, e.g., the read-copy update mechanism in Linux.

For certain application domains, virtualized systems can provide good scalability at relatively low engineering costs. Virtualization allows the uniprocessor kernel to be kept unchanged for fully virtualized environments or with small changes in paravirtualized environments. For example, the Xen hypervisor implementation is very small compared to Linux, and the changes needed to port an operating system is fairly limited. However, hardware support is needed for fault tolerance, and shared-memory applications cannot be load-balanced across virtual machines.

Reimplementation allows the highest scalability improvements but at the highest effort. Reimplementation should mainly be considered if the original operating system would be very hard to port with good results, or if the target hardware is very different from the current platform.

The Linux case study illustrates the evolution of multiprocessor support for a kernel. The 2.0 giant lock implementation was kept close to the uniprocessor. The implementation then adopted a more coarse-grained locking approach, which became significantly more complex and also diverged more from the uniprocessor kernel. The more fine-grained approaches in 2.4 and 2.6 do not increase the complexity as compared to 2.2, which suggests that the implementation converges again as it matures.

Acknowledgments

This work was partly funded by The Knowledge Foundation in Sweden under a research grant for the project “Blekinge - Engineering Software Qualities (BESQ)” (<http://www.ipd.bth.se/~besq>).

References

- [1] J. Appavoo, M. Auslander, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, J. Xenidis, M. Stumm, B. Gamsa, R. Azimi, R. Fingas, A. Tam, and D. Tam. Enabling Scalable Performance for General Purpose Workloads on Shared Memory Multiprocessors. Technical Report IBM Research Report RC22863, IBM Research, 2003.
- [2] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.
- [3] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 2nd edition, 1998.
- [4] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *OSDI*, pages 179–193, Nov. 1994.
- [5] R. Clark, J. O’Quin, and T. Weaver. Symmetric multiprocessing for the AIX operating system. In *Compton ’95*, pages 110–115, 1995.
- [6] J. M. Denham, P. Long, and J. A. Woodward. DEC OSF/1 symmetric multiprocessing. *Digital Technical Journal*, 6(3), 1994.
- [7] S. Doherty, D. L. Detlefs, L. Grove, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and J. Guy L. Steele. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA ’04*, pages 216–224, 2004.
- [8] L. G. Gerbarg. Advanced synchronization in Mac OS X: Extending UNIX to SMP and real-time. In *Proceedings of BSDCon 2002*, pages 37–45, Feb 2002.
- [9] G. H. Goble and M. H. Marsh. A dual processor VAX 11/780. In *ISCA ’82*, pages 291–298, 1982.
- [10] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP’99*, pages 154–169, December 1999.

- [11] J. Katcher. Postmark: A new file system benchmark. Technical Report Technical Report TR3022, Network Appliance. www.netapp.com/tech_library/3022.html.
- [12] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric multiprocessing in Solaris 2.0. In *Compcon*, pages 181–186, 1992.
- [13] S. Kågström, H. Grahn, and L. Lundberg. Experiences from implementing multiprocessor support for an industrial operating system kernel. In *RTCSA '2005 (to appear)*, August 2005.
- [14] S. Kågström, L. Lundberg, and H. Grahn. A novel method for adding multiprocessor support to a large and complex uniprocessor kernel. In *IPDPS 2004*, April 2004.
- [15] G. Lehey. Improving the FreeBSD SMP implementation - a case study. In *Asian Enterprise Open Source Conference*, October 2003.
- [16] R. Love. *Linux Kernel Development*. Sams, Indianapolis, Indiana, 1st edition, 2003.
- [17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [18] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):8, 1992.
- [19] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.
- [20] S. J. Muir. *Piglet: an operating system for network appliances*. PhD thesis, University of Pennsylvania, 2001.
- [21] B. Mukherjee, K. Schwan, and P. Gopinath. A Survey of Multiprocessor Operating System Kernels. Technical Report GIT-CC-92/05, Georgia Institute of Technology, 1993.
- [22] QNX Software Systems Ltd. The QNX Neutrino microkernel, 2003. <http://qdn.qnx.com>.

- [23] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38(5):39–47, May 2005.
- [24] C. Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, Boston, 1st edition, 1994.
- [25] J. Talbot. Turning the AIX operating system into an MP-capable OS. In *Proceedings of the 1995 USENIX Annual Technical Conference*, Jan. 1995.
- [26] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, May 6–7 2004.
- [27] K. Yaghmour. A practical approach to Linux clusters on SMP hardware, <http://www.opersys.com>, July 2002.