

The Design and Implementation of Multiprocessor Support for an Industrial Operating System Kernel*

Simon Kågström, Håkan Grahn, and Lars Lundberg

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
P.O. Box 520, SE-372 25 Ronneby, Sweden
{ska, hgr, llu}@bth.se

Abstract

The ongoing transition from uniprocessor to multiprocessor computers requires support from the operating system kernel. Although many general-purpose multiprocessor operating systems exist, there is a large number of specialized operating systems which require porting in order to work on multiprocessors. In this paper we describe the multiprocessor port of a cluster operating system kernel from a producer of industrial systems. Our initial implementation uses a giant locking scheme that serializes kernel execution. We also employed a method in which CPU-local variables are placed in a special section mapped to per-CPU physical memory pages. The giant lock and CPU-local section allowed us to implement an initial working version with only minor changes to the original code, although the giant lock and kernel-bound applications limit the performance of our multiprocessor port. Finally, we also discuss experiences from the implementation.

Keywords: Multiprocessor, operating system kernel, implementation experiences

1 Introduction

A current trend in the computer industry is the transition from uniprocessors to various kinds of multiprocessors, also for desktop and embedded systems. Apart from traditional SMP systems, many manufacturers are now presenting chip multiprocessors or simultaneous multithreaded CPUs [8, 15, 20] which allow more efficient use of chip area. The trend towards multiprocessors requires support from operating systems and applications to take advantage of the hardware.

While there are many general-purpose operating systems for multiprocessor hardware, it is not always possible to adapt special-purpose applications to run on these operating systems, for example due to different programming models. These applications often rely on support from customized operating systems, which frequently run on uniprocessor hardware. There are many important application areas where this is the case, for example in telecommunication systems or embedded systems. To benefit from the new hardware, these operating systems must be adapted.

We are working on a project together with a producer of large industrial systems in providing multiprocessor support for an operating system kernel. The operating system is a special-purpose industrial system primarily used in telecommunication systems. It currently runs on clusters of uniprocessor Intel IA-32 computers, and provides high availability and fault tolerance as well as (soft) real-time response time and high throughput performance. The system can run on one of two operating system kernels, either the Linux kernel or an in-house kernel, which is an object-oriented operating system kernel implemented in C++. The

*This is an extended version of the paper "Experiences from Implementing Multiprocessor Support for an Industrial Operating System Kernel" published on RTCSA-2005 [11]

in-house kernel offers higher performance while Linux provides compatibility with third-party libraries and tools. With multiprocessor hardware becoming cheaper and more cost-effective, a port to multiprocessor hardware is becoming increasingly interesting to harvest the performance benefits of the in-house kernel.

In this paper, we describe the design and implementation of initial multiprocessor support for the in-house kernel. We have also conducted a set of benchmarks to evaluate the performance, and also profiled the locking scheme used in our implementation. Some structure names and terms have been modified to keep the anonymity of our industrial partner.

The rest of the paper is structured as follows. Section 2 describes the structure and the programming model for the operating system. Section 3 thereafter describes the design decisions made for the added multiprocessor support. Section 4 outlines the method we used for evaluating our implementation, and Section 5 describes the evaluation results. We thereafter discuss some experiences we made during the implementation in Section 6 and describe related and future work in Section 7. Finally, we conclude in Section 8.

2 The Operating System

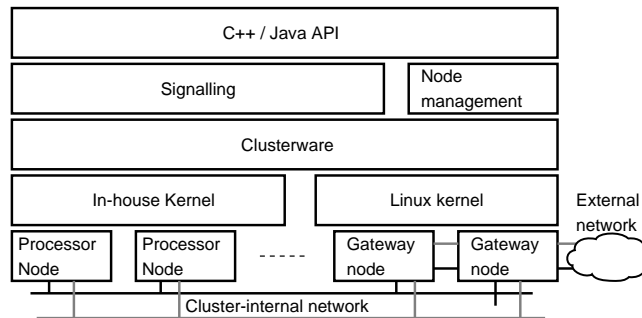


Figure 1: The architecture of the operating system.

Figure 1 shows the architecture of the operating system. The system exports a C++ or Java API to application programmers for the clusterware. The clusterware runs on top of either the in-house kernel or Linux and provides access to a distributed RAM-resident database, cluster management that provides fail-safe operation and an object broker (CORBA) that provides interoperability with other systems.

A cluster consists of processing nodes and gateway machines. The processing nodes handle the workload and usually run the in-house kernel. Gateway machines run Linux and act as front-ends to the cluster, forwarding traffic to and from the cluster. The gateway machines further provide logging support for the cluster nodes and do regular backups to hard disk of the database. The cluster is connected by redundant Ethernet connections internally, while the connections to the outside world can be either SS7 [7] or Ethernet. Booting a node is performed completely over the network by PXE [6] and TFTP [19] requests handled by the gateway machines.

2.1 The Programming Model

The operating system employs an asynchronous programming model and allows application development in C++ and Java. The execution is event-based and driven by callback functions invoked on events such as inter-process communication, process startup, termination, or software upgrades. The order of calling the functions is not specified and the developer must adapt to this. However, the process will be allowed to finish execution of the callbacks before being preempted, so two callbacks will never execute concurrently in one process.

In the operating system, two types of processes, *static* and *dynamic*, are defined. Static processes are restarted on failure and can either be unique or replicated in the system. For unique static processes, there is only one process of that type in the whole system, whereas for replicated processes, there is one process

per node in the system. If the node where a unique process resides crashes, the process will be restarted on another node in the system. Replicated static processes allow other processes to communicate with the static process on the local node, which saves communication costs.

Dynamic processes are created when referenced by another process, for example by a static process. The dynamic processes usually run short jobs, for instance checking and updating an entry in the database. Dynamic processes are often tied to database objects on the local node to provide fast access to database objects. In a telecommunication billing system for example, a static process could be used to handle new calls. For each call, the static process creates a dynamic process, which, in turn, checks and updates the billing information in the database.

2.2 The Distributed Main-Memory Database

The operating system employs an object-oriented distributed RAM-resident database which provides high performance and fail-safe operation. The database stores persistent objects which contain data and have methods just like other objects. The objects can be accessed transparently across nodes, but local objects are faster to access than remote ones (which is the reason to tie processes to database objects).

For protection against failures, each database object is replicated on at least two nodes. On hardware or software failure, the cluster is reconfigured and the database objects are distributed to other nodes in the cluster.

2.3 The Process and Memory Model

The operating system base user programs on three basic entities: *threads*, *processes*, and *containers*. The in-house kernel has kernel-level support for threading, and threads define the basic unit of execution for the in-house kernel. Processes act as resource holders, containing open files, sockets, etc., as well as one or more threads. Containers, finally, define the protection domain (an address space). Contrary to the traditional UNIX model, the in-house kernel separates the concepts of address space and process, and a container can contain one or more processes, although there normally is a one-to-one correspondence between containers and processes.

To allow for the asynchronous programming model and short-lived processes, the in-house kernel supplies very fast creation and termination of processes. There are several mechanisms behind the fast process handling. First, each code package (object code) is located at a unique virtual address range in the address space. All code packages also reside in memory at all times, i.e., similar to single-address space operating systems [2, 5]. This allows fast setup of new containers since no new memory mappings are needed for object code. The shared mappings for code further means that there will never be any page faults on application code, and also that RPC can be implemented efficiently.

The paging system uses a two-level paging structure on the IA-32. The first level on the IA-32 is called a *page directory* and is an array of 1024 *page directory entries*, each pointing to a *page table* mapping 4MB of the address space. Each *page table* in turn contains *page table entries* which describe the mapping of 4KB virtual memory pages to physical memory pages. During kernel initialization, a global page directory containing application code and kernel code and kernel data is created, and this page directory then serves as the basis for subsequent page directories since most of the address space is identical between containers. The address space of the in-house kernel is shown in Figure 2.

The in-house kernel also keeps all data in-memory at all times, so there is no overhead for handling pageout to disk. Apart from reducing time spent in waiting for I/O, this also reduces the complexity of page fault handling. A page fault will never cause the faulting thread to sleep, and this simplifies the page fault handler and improves real-time predictability of the page fault latency.

The memory allocated to a container initially is very small. The container process (which will be single-threaded at startup time), starts with only two memory pages allocated, one containing the page table and the other the first 4KB of the process stack. Because of this, the container can use the global page directory, replacing the page directory entry for the 4MB region which contains the entire container stack, the global variables, and part of the heap. Any page fault occurring in this 4MB region can be handled by adding pages to the page table. For some processes, this is enough, and they can run completely in the global page directory.

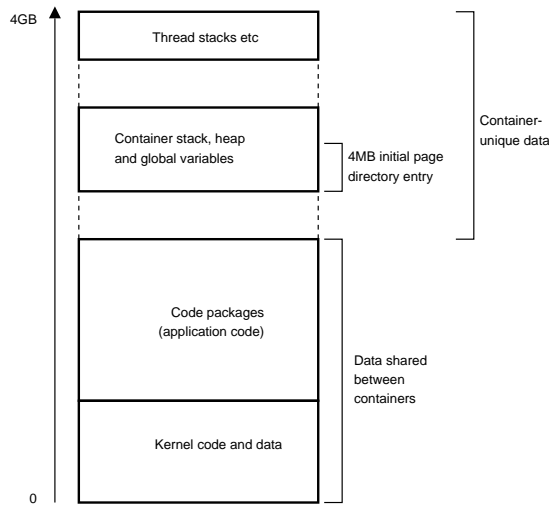


Figure 2: The in-house kernel address space on Intel IA-32 (simplified).

Figure 3 shows the container address space handling in the operating system. In Figure 3a, the situation right after process startup is shown. The container first uses the global page directory, with two pages allocated: one for the stack page table and one for the process stack. This situation gradually evolves into Figure 3b, where the process has allocated more pages for the stack, the heap or global variables, still within the 4MB area covered by the stack page table. When the process accesses data outside the stack page table, the global page directory can no longer be used and a new page directory is allocated and copied from the global as shown in Figure 3c.

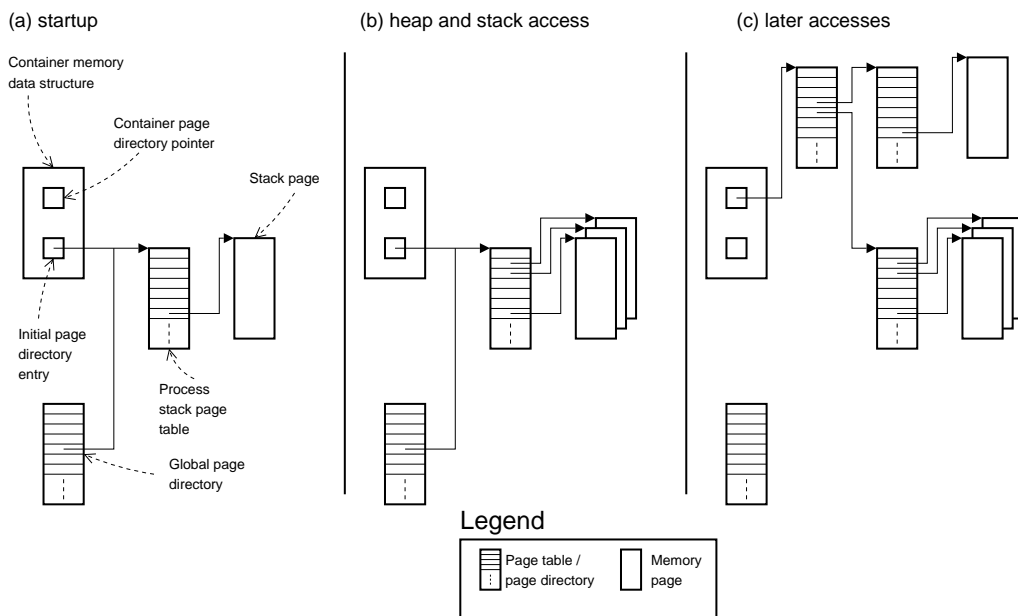


Figure 3: Handling of container address spaces in the in-house kernel

3 Design of the Multiprocessor Support

In this section, we discuss the design of multiprocessor support for the in-house kernel. We describe the locking scheme we adopted, the implementation of CPU-local data, and optimizations made possible by the special properties of the in-house kernel.

3.1 Kernel Locking and Scheduling

For the first multiprocessor implementation, we employ a simple locking scheme where the entire kernel is protected by a single, “giant” lock (see Chapter 10 in [18]). The giant lock is acquired when the kernel is entered and released again on kernel exit. The advantage of the giant locking mechanism is that the implementation is kept close to the uniprocessor version. Using the giant lock, the uniprocessor semantics of the kernel can be kept, since two CPUs will never execute concurrently in the kernel. For the initial version, we deemed this important for correctness reasons and to get a working version early. However, the giant lock has shortcomings in performance since it locks larger areas than potentially needed. This is especially important for kernel-bound processes and multiprocessors with many CPUs. Later on, we will therefore relax the locking scheme to allow concurrent access to parts of the kernel.

We also implemented CPU-affinity for threads in order to avoid cache lines being moved between processors. Since the programming model in the operating system is based on short-lived processes, we chose a model where a thread is never migrated from the CPU it was started on. For short-lived processes, the cost of migrating cache lines between processors would cause major additional latency. Further, load imbalance will soon even out with many short processes. With fast process turnaround, newly created processes can be directed to idle CPUs to quickly even out load imbalance.

3.2 CPU-local Data

Some structures in the kernel need to be accessed privately by each CPU. For example, the currently running thread, the current address space, and the kernel stack must be local to each CPU. A straightforward method of solving this would be to convert the affected structures into vectors, and index them with the CPU identifier. However, this would require extensive changes to the kernel code, replacing every access to the structure with an index-lookup. It would also require three more instructions (on IA-32) for every access, not counting extra register spills etc.

This led us to adapt another approach instead, where each CPU always runs in a private address space. With this approach, each CPU accesses the CPU-local data at the same virtual address without any modifications to the code, i.e., access of a CPU-local variable is done exactly as in the uniprocessor kernel. To achieve this, we reserve a 4KB virtual address range for CPU-local data and map this page to different physical pages for each CPU. The declarations of CPU-local variables and structures are modified to place the structure in a special ELF-section [21], which is page-aligned by the boot loader.

The CPU-local page approach presents a few problems, however. First, some CPU-local structures are too large to fit in one page of memory. Second, handling of multithreaded processes must be modified for the CPU-local page, which is explained in the next section. The kernel stack, which is 128KB per CPU, is one example of a structure which is too large to store in the CPU-local page. The address of the kernel stack is only needed at a few places, however, so we added a level of indirection to set the stack pointer register through a CPU-local pointer to the kernel stack top. The global page directory (which needs to be per-CPU since it contains the CPU-local page mapping) is handled in the same manner.

3.3 Multithreaded Processes

The CPU-local page presents a problem for multithreaded containers (address spaces). Normally, these would run in the same address space, which is no problem on a uniprocessor system. In a multiprocessor system, however, using a single address space for all CPUs would cause the CPU-local virtual page to map to the same physical page for all CPUs, i.e., the CPU-local variables would be the same for all CPUs. To solve this problem, a multithreaded container needs a separate page directory for every CPU which

executes threads in the container. However, we do not want to compromise the low memory requirements for containers by preallocating a page for every CPU.

Since multithreaded containers are fairly rare in the operating system, we chose a lazy method for handling the CPU-local page in multithreaded containers. Our method allows singlethreaded containers to run with the same memory requirements as before, while multithreaded containers require one extra memory page per CPU which executes in the container. Further, the method requires only small modifications to the kernel source code and allows for processor affinity optimizations without changes.

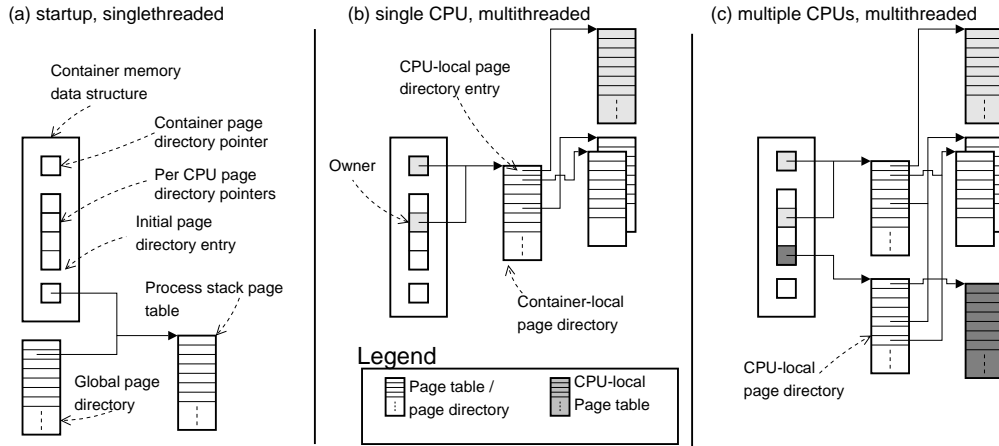


Figure 4: Handling of container address spaces in the in-house kernel for multiprocessor computers

Figure 4 shows the handling of multithreaded containers on multiprocessors in the in-house kernel. The figure shows the container memory data structure, which has a container page directory pointer and an initial page directory entry as before (see Figure 3 and Section 2.3), but has also been extended with an array of per-CPU page directory pointers.

When the process starts up it will have only one thread and the situation is then as in Figure 4a. The process initially starts without a private address space and instead uses the global address space (which is CPU-local). The global page directory is modified with a page table for the process stack, global variables and part of the heap. As long as the process is singlethreaded and uses moderate amounts of heap or stack space, this will continue to be the case.

When the process becomes multithreaded the first time, as shown in Figure 4b, a new *container page directory* is allocated and copied from the global page directory¹. The current CPU will then be set as the owner of the container page directory. The CPU-local entry of the page directory is thereafter setup to point to the CPU-local page table of the CPU that owns the container page directory. Apart from setting the owner, this step works exactly as in the uniprocessor version. Since the thread stacks reside outside the 4MB process stack area, multithreaded processes will soon need a private address space, so there is no additional penalty in setting up the address space immediately when the process becomes multithreaded.

As long as only one CPU executes the threads in the process, there will be only one page directory used. However, as soon as another CPU schedules a thread in the process, a single page directory is no longer safe. Therefore, the container page directory is copied to a new CPU-local page directory which is setup to map the CPU-local page table. This is shown in Figure 4c. Note that apart from the CPU-local page table, all other page tables are identical between the CPUs. When scheduling the thread, the CPU-local page directory will be used.

One complication with this scheme is page fault handling. If two or more CPUs run in a container, a page fault will be generated for the CPU-local page directory. We therefore modified the page fault handler to always update the container page directory beside the CPU-local page directory. However, there can still be inconsistencies between page directories if the owner of the container page directory causes a page fault,

¹Note that a new page directory can be allocated for singlethreaded processes as well, if they access memory outside the 4MB area of the stack page table.

which would only update the container page directory. A later access on the same page from another CPU will then cause a spurious page fault. We handle this situation lazily by checking if the page was already mapped in the container page directory, in which case we just copy the entry to the faulting page directory. Note that this situation is fairly uncommon since it only affects faults on unmapped page directories, i.e., 4MB areas. Faults on 4KB pages will be handled transparently of our modifications since the page tables are shared by all CPUs.

We also handle inconsistencies in the address translation cache (TLB) lazily. If a page table entry in a container is updated on one CPU, the TLBs on other CPUs executing in the container can contain stale mappings, which is another source of spurious page faults. Spurious page faults from a inconsistent TLB can be safely ignored in the in-house kernel since pages are never unmapped from a container while the process is running. This saves us from invalidating the TLBs on other CPUs, which would otherwise require an inter-processor interrupt.

4 Evaluation Framework

We have performed an initial evaluation of our multiprocessor implementation where we evaluate contention on our locking scheme as well as the performance of the multiprocessor port. We ran all performance measurements on a two-way 300MHz Pentium II SMP equipped with 128MB SDRAM main memory.

For the performance evaluation, we constructed a benchmark application which consists of two processes executing a loop in user-space which at configurable intervals performs a kernel call. We then measured the time needed (in CPU-cycles) to finish both of these processes. This allows us to vary the proportion of user to kernel execution, which will set the scalability limit for the giant locking approach. Unfortunately we were not able to configure the operating system to run the benchmark application in isolation, but had to run a number of system processes beside the benchmark application. This is incorporated into the build process for applications, which normally need support for database replication, logging etc. During the execution of the benchmark, around 100 threads were started in the system (although not all were active).

We also benchmarked the locking scheme to see the proportion of time spent in holding the giant lock, spinning for the lock, and executing without the lock (i.e., executing user-level code). The locking scheme was benchmarked by instrumenting the acquire lock and release lock procedures with a reading of the CPU cycle counter. The lock time measurement operates for one CPU at a time, in order to avoid inconsistent cycle counts between the CPUs and to lessen the perturbation from the instrumentation on the benchmark. The locking scheme is measured from the start of the benchmark application until it finishes.

5 Evaluation Results

In this section we present the evaluation results for the locking scheme and the application benchmark. We also evaluate our CPU-affinity optimization and the slowdown of running the multiprocessor version of the operating system on a uniprocessor machine. Consistent speedups are only seen when our benchmark application executes almost completely in user-mode, so the presented results refer to the case when the benchmark processes run only in user-mode.

Executing the benchmark with the multiprocessor kernel on a uniprocessor gives a modest slowdown of around 2%, which suggests that our implementation has comparatively low overhead and that the multiprocessor kernel can be used even on uniprocessor hardware. Running the benchmark on the multiprocessor gives a 20% speedup over the uniprocessor kernel, which was less than we expected. Since the two benchmark processes run completely in user-mode and does not interact with each other, we expected a speedup close to 2.0 (slightly less because of interrupt handling costs etc.).

Table 1 shows the lock contention when the benchmark application run completely in user-mode, both the uniprocessor and the multiprocessor. For the uniprocessor, acquiring the lock always succeeds immediately. From the table, we can see that the uniprocessor spends around 36% of the time in the kernel. On the multiprocessor, all times are shared between two CPUs, and we see that 20%-23% of the time is

Table 1: Proportion of time spent executing user and kernel code.

	User-mode	Kernel	Spinning
UP	64%	36%	< 0.1%
SMP	55%-59%	20%-22%	20-23%

spent spinning for the giant lock. Since the in-kernel time is completely serialized by the giant lock, the theoretically maximum speedup we can achieve on a dual processor system is $\frac{36+64}{36+\frac{64}{2}} \approx 1.47$ according to Amdahl's law.

There are several reasons why the speedup is only 1.2 for our benchmark. First, the benchmark processes do not execute in isolation, which increases the in-kernel time and consequently the time spent spinning for the lock. Second, some heavily accessed shared data structures in the kernel, e.g., the ready queue cause cache lines to be transferred between processors, and third, spinning on the giant lock effectively makes the time spent in-kernel on the multiprocessor longer than for the uniprocessor.

CPU-affinity does not exhibit clear performance benefits, with the benchmark finishing within a few percent faster than without affinity. This is likely caused because of the high proportion of in-kernel execution. We also tried some other optimizations such as prioritizing the benchmark processes over other processes and different time slice lengths, but did not get any significant benefits over the basic case.

6 Implementation Experiences

The implementation of multiprocessor support for the in-house kernel was more time consuming than we had first expected. The project has been ongoing part-time for two years, during which a single developer has performed the multiprocessor implementation. Initially, we expected that a first version would be finished much sooner, in approximately six months. The reasons for the delay are manifold.

First, the development of a multiprocessor kernel is generally harder than a uniprocessor kernel because of inherent mutual exclusion issues. We therefore wanted to perform the development in the Simics full-system simulator [14], and a related project investigated running the operating system on Simics. It turned out, however, that it was not possible at that time to boot the system on Simics because of lacking hardware support in Simics. Second, we performed most of the implementation off-site, which made it harder to get assistance from the core developers. Coupled to the fact that the system is highly specialized and complex to build and setup, this led us to spend significant amount of time on configuration issues and build problems. Finally, the code base of the operating system is large and complex. The system consists of over 2.5 million lines totally, of which around 160,000 were relevant for our purposes. The complexity and volume of the code meant that we had to spend a lot of time to grasp the functionality of the code.

In the end, we wrote around 2,300 lines of code in new files and modified 1,600 existing lines for the implementation. The new code implement processor startup and support for the locking scheme whereas the modified lines implement CPU-local data, acquiring and releasing the giant lock etc. The changes to the original code is limited to around 1% of the total relevant code base, which shows that it is possible to implement working multiprocessor support with a relatively modest engineering effort. We chose the simple giant lock to get a working version fast and the focus is now on continuous improvements which we discuss in Section 7.

7 Related and Future Work

The operating system studied in this paper has, as mentioned before, a number of properties that are different from other cluster operating systems. It provides a general platform with high availability and high performance for distributed applications and an event-oriented programming environment based on fast process handling. Most other platforms/programming environments are mainly targeted at high performance and/or parallel and distributed programming, e.g., MPI [16] or OpenMP [17]. These systems run on networked computer nodes running a standard operating system, and are not considered as cluster operating systems.

There exists some distributed operating systems running on clusters of Intel hardware. One such example is Plurix [4], which has several similarities with the operating system. Plurix provides a distributed shared memory where communication is done through shared objects. The consistency model in Plurix is based on restartable transactions coupled with an optimistic synchronization scheme. The distributed main memory database in the operating system serves the same purpose. However, to the best of our knowledge, Plurix only runs on uniprocessor nodes and not on multiprocessors in a cluster. Plurix is also Java-based whereas the operating system presented in this paper supports both C++ and Java development.

Many traditional multiprocessor operating systems have evolved from monolithic uniprocessor kernels, e.g., Linux and BSD. Such monolithic kernels contain large parts of the actual operating system which make multiprocessor adaptation a complex task. Early multiprocessor operating systems often used coarse-grained locking, for example using a giant lock [18]. The main advantage with the coarse-grained method is that most data structures of the kernel can remain unprotected, and this simplifies the multiprocessor implementation. For example, Linux and FreeBSD both initially implemented giant locks [1, 12].

For systems which have much in-kernel time, the time spent waiting for the kernel lock can be substantial, and in many cases actually unnecessary since the processors might use different paths through the kernel. Most evolving multiprocessor kernels therefore moves toward finer-grained locks. The FreeBSD multiprocessor implementation has for example shifted toward a fine-grained method [12] and mature UNIX systems such as AIX and Solaris implement multiprocessor support with fine-grained locking [3, 9], as do current versions of Linux [13].

Like systems which use coarse-grained locking, master-slave systems (refer to Chapter 9 in [18]) allow only one processor in the kernel at a time. The difference is that in master-slave systems, one processor is dedicated to handling kernel operations (the “master” processor) whereas the other processors (“slave” processors) run user-level applications and only access the kernel indirectly through the master processor. Since all kernel access is handled by one processor, this method limits throughput for kernel-bound applications.

In [10], an alternative porting approach focusing on implementation complexity is presented. The authors describe the *application kernel approach*, whereby the original uniprocessor kernel is kept as-is and the multiprocessor support is added as a loadable module to the uniprocessor kernel. This allows the uniprocessor kernel to remain essentially unchanged, avoiding the complexity of in-kernel modifications. The approach is similar to master-slave systems performance-wise since all kernel operations are performed by one processor in the system. Neither the master-slave approach nor the application kernel approach provide any additional performance benefit over our giant lock, and incrementally improving the giant locking with finer-grained strategies is easier.

The in-house kernel uses a large monolithic design. The kernel contains very much functionality such as a distributed fault-tolerant main-memory database and support for data replication between nodes. Therefore, adding multiprocessor support is a very complex and challenging task. In the operating system, a large portion of the execution time is spent in the kernel, making it even more critical when porting the kernel to multiprocessor hardware. As described earlier in this paper we chose a giant lock solution for our first multiprocessor version of the in-house kernel in order to get a working version with low engineering effort. As a result of the single kernel-lock and the large portion of kernel time, this locking strategy resulted in rather poor multiprocessor performance.

Future work related to the multiprocessor port of the in-house kernel will be focused around the following. The speedup is low when running on more than one CPU because of the giant lock and kernel-bound applications. Therefore, one of our next steps is to implement a more fine-grained locking structure. As an example, we are planning to use a separate lock for low-level interrupt handling to get lower interrupt latency. Further, we will also identify the parts of the kernel where the processor spend most time, which could be good candidates for subsystem locks. Another area of possible improvements is the CPU scheduler where we will investigate dividing the common ready queue into one queue per processor, which is done in for example Linux 2.6 [13].

Finally, we would like to further explore CPU-affinity optimizations for short-lived processes. For example, although the processes currently will not move to another processor, it might be started on another processor the next time it is created. Depending on the load on the instruction cache, keeping later processes on the same processor might be beneficial by avoiding pollution of the instruction caches.

8 Conclusions

In this paper, we have described the design decisions behind an initial multiprocessor port of an in-house cluster operating system kernel. The in-house kernel is a high performance fault-tolerant operating system kernel targeted at soft real-time telecommunication applications.

Since our focus was to get an initial version with low engineering effort, we chose a simple “giant” locking scheme where a single lock protects the entire kernel from concurrent access. The giant locking scheme allowed us to get a working version without making major changes to the uniprocessor kernel, but it has some limitations in terms of performance. Our model where CPU-local variables are placed in a virtual address range mapped to unique physical pages on different CPUs allowed us to keep most accesses of private variables unchanged. We also show how this method can be applied to multithreaded processes with a very small additional memory penalty.

The evaluation we made shows that there is room for performance improvements, mainly by relaxing the locking scheme to allow concurrent kernel execution. Our experience illustrates that refactoring of a large and complex industrial uniprocessor kernel for multiprocessor operation is a major undertaking, but also that it is possible to implement multiprocessor support without intrusive changes to the original kernel (only changing around 1% of the core parts of the kernel).

Acknowledgments

The authors would like to thank the PAARTS-group at BTH for valuable comments and ideas for the implementation and paper. 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.bth.se/-besq>).

References

- [1] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 2nd edition, 1998.
- [2] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. Technical Report TR-92-03-02, 1992.
- [3] R. Clark, J. O’Quin, and T. Weaver. Symmetric multiprocessing for the AIX operating system. In *Compton ’95. Technologies for the Information Superhighway’, Digest of Papers.*, pages 110–115, 1995.
- [4] R. Goeckelmann, M. Schoettner, S. Frenz, and P. Schulthess. A kernel running in a DSM – design aspects of a distributed operating system. In *IEEE International Conference on Cluster Computing (CLUSTER’03)*, pages 478–482. IEEE, December 2003.
- [5] Gernot Heiser, Kevin Elphinstone, Jerry Vochtelloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9):901–928, 1998.
- [6] Intel Corporation. *Preboot Execution Environment (PXE) Specification*, September 1999. Version 2.1.
- [7] ITU-T. *ITU-T Recommendation Q.700, Introduction To ITU-T Signalling System No. 7 (SS7)*. International Telecommunication Union, 1993.
- [8] J. Kahle. Power4: A dual-CPU processor chip. In *Proceedings of the 1999 International Microprocessor*, San Jose, CA, October 1999.
- [9] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric multiprocessing in Solaris 2.0. In *Compton*, pages 181–186. IEEE, 1992.

- [10] S. Kågström, L. Lundberg, and H. Grahn. A novel method for adding multiprocessor support to a large and complex uniprocessor kernel. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, USA, April 2004.
- [11] Simon Kågström, Håkan Grahn, and Lars Lundberg. Experiences from implementing multiprocessor support for an industrial operating system kernel. In *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'2005)*, pages 365–368, Hong Kong, China, August 2005.
- [12] G. Lehey. Improving the FreeBSD SMP implementation - a case study. In *Asian Enterprise Open Source Conference*, Singapore, October 2003.
- [13] Robert Love. *Linux Kernel Development*. Sams, Indianapolis, Indiana, 1st edition, 2003.
- [14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [15] D.T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.
- [16] Message Passing Interface (MPI) Forum. *The Message Passing Interface Standard*. See <http://www.mpi-forum.org/>, accessed 28/7-2005.
- [17] OpenMP Architecture Review Board. *OpenMP Version 2.5 Specification*, May 2005. See <http://www.openmp.org>, accessed 28/7-2005.
- [18] C. Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, Boston, 1st edition, 1994.
- [19] K. Sollins. *The TFTP Protocol (Revision 2) (RFC 1350)*. MIT, July 1992. STD 33.
- [20] Sun Microsystems. *Sun UltraSPARC IV*, February 2004. See <http://www.sun.com/processors/-UltraSPARC-IV/>, accessed 28/7-2005.
- [21] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification*, 1995. Version 1.2.