

The Application Kernel Approach - a Novel Approach for Adding SMP Support to Uniprocessor Operating Systems*

Simon Kågström, Håkan Grahn, 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

December 21, 2005

Abstract

The current trend of using multiprocessor computers for server applications require operating system adaptations to take advantage of more powerful hardware. However, modifying large bodies of software is very costly and time-consuming, and the cost of porting an operating system to a multiprocessor might not be motivated by the potential performance benefits. In this paper we present a novel method, the *application kernel approach*, for adaption of an existing uniprocessor kernel to SMP hardware. Our approach considers the existing uniprocessor kernel as a “black box”, to which no or very small changes are made. Instead, the original kernel runs OS-services unmodified on one processor whereas the other processors execute applications on top of a small custom kernel. We have implemented the application kernel for the Linux operating system, which illustrates that the approach can be realized with fairly small resources. We also present an evaluation of the performance and complexity of our approach, where we show that it is possible to achieve good performance while at the same time keeping the implementation complexity low.

1 Introduction

For performance reasons, uniprocessor computers are now being replaced with small multiprocessors. Moreover, modern processor chips from major processor manufacturers often contain more

*An earlier and less complete version of this paper was published at the 18th International Parallel and Distributed Processing Symposium (IPDPS) 2004.

than one CPU core, either logically through Symmetric MultiThreading [11] or physically as a Chip MultiProcessor [17]. For instance, current Intel Pentium 4 and Xeon processors contain two logical processors [29] and several other manufacturers are in the process of introducing on-chip multiprocessors [20,37]. With multiprocessors becoming prevalent, good operating system support is crucial to benefit from the increased computing capacity.

We are currently working on a project together with a major developer of industrial systems. The company has over the last 10 years been developing an operating system kernel for clusters of uniprocessor IA-32 computers. The operating system has interesting properties such as fault tolerance and high performance (mainly in terms of throughput). In order to take advantage of new shared-memory multiprocessors, a multiprocessor version of the kernel is being developed [24]. However, we were faced with the problem that it was very difficult and costly to make the needed modifications because of the size of the code, the long time during which the code had been developed (this has led to a code structure which is hard to grasp), and the intricate nature of operating system kernels.

The situation described above illustrates the fact that making changes to large software bodies can be very costly and time consuming, and there has also been a surge of interest in alternative methods lately. For example, as an alternative to altering operating system code, Arpaci-Dusseau et al. [1] propose a method where “gray-box” knowledge about algorithms and the behavior of an operating system are used to acquire control and information over the operating system without explicit interfaces or operating system modification. There has also been some work where the kernel is changed to provide quality of service guarantees to large unmodified applications [41].

For the kernel of our industrial partner, it turned out that the software engineering problems when adding multiprocessor support were extremely difficult and time-consuming to address using a traditional approach. Coupled to the fact that the target hardware would not scale to a very large number of processors during the foreseeable future (we expect systems in the range of 2 to 8 processors), this led us to think of another approach. In our approach, we treat the existing kernel as a black box and build the multiprocessor adaptations beside it. A custom kernel called *the application kernel*, of which the original kernel is unaware, is constructed to run on the other processors in the system while the original kernel continues to run on the boot processor. Applications execute on the other processors while system calls, page faults, etc., are redirected by the application kernel to the uniprocessor kernel. We expect the application kernel approach to substantially lower the development and maintenance costs compared to a traditional multiprocessor port.

In this paper, we describe the application kernel approach and evaluate an implementation for the Linux kernel. With this implementation, we demonstrate that it is possible to implement our approach without changing the kernel source code and at the same time running unmodified Linux applications. We evaluate our approach both in terms of performance and implementation complexity. The evaluation results show that the implementation complexity is low in terms of lines of code and cyclomatic complexity for functions, requiring only seven weeks to implement. Performance-wise, our implementation performance-levels comparable to Linux for compute-bound applications.

The application kernel implementation for Linux is available as free software licensed under the GNU GPL at http://www.ipd.bth.se/ska/application_kernel.html. This paper builds on our previous work where we implemented the application kernel approach for a small in-house kernel [23].

The rest of the paper is structured as follows. We begin with discussing related work in Section 2. In Section 3 we describe the ideas behind our approach and Section 4 then discusses our implementation for the Linux kernel. We describe our evaluation framework in Section 5, and then evaluate the implementation complexity and performance of the application kernel in Section 6. Finally, we conclude and discuss future extensions to the approach in Section 7.

2 Related Work

The implementation of a multiprocessor operating system kernel can be structured in a number of ways. In this section, we present the traditional approaches to multiprocessor porting as well as some alternative methods and discuss their relation to our approach.

2.1 Monolithic Kernels

Many multiprocessor operating systems have evolved from monolithic uniprocessor kernels. These uniprocessor kernels (for example Linux and BSD UNIX) contain large parts of the actual operating system, making multiprocessor adaptation a complex task. In-kernel data structures need to be protected from concurrent access from multiple processors and this requires locking. The granularity of the locks, i.e., the scope of the code or data structures a lock protects, is an important component for the performance and complexity of the operating system. Early multiprocessor operating systems often used coarse-grained locking, for example the semaphore-based multiprocessor version of UNIX described by Bach and Buroff [2]. These systems employ a locking scheme where only one processor runs in the kernel (or in a kernel subsystem) at a time [36]. 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. In the most extreme case, a single “giant” lock protects the entire kernel.

The time spent in waiting for the kernel locks can be substantial for systems dominated by in-kernel execution, and in many cases actually unnecessary since the processors might use different paths through the kernel. The obvious alternative is then to relax the locking scheme and use a more fine grained locking scheme to allow several processors to execute in the kernel concurrently. Fine-grained systems allow for better scalability since processes can run with less blocking on kernel-access. However, they also require more careful implementation, since more places in the kernel must be locked. The FreeBSD SMP implementation, which originally used coarse-grained locking, has shifted toward a fine-grained method [25] and mature UNIX systems such as AIX and Solaris implement multiprocessor support with fine-grained locking [8, 21], as do current versions of Linux [27].

2.2 Microkernel-based Systems

Another approach is to run the operating system on top of a microkernel. Microkernel-based systems potentially provide better system security by isolating operating system components and also better portability since much of the hardware dependencies can be abstracted away by the microkernel. There are a number of operating systems based on microkernels, e.g., L4Linux [18], a modified Linux kernel which runs on top of the L4 microkernel [26]. The Mach microkernel has been used as the base for many operating systems, for example DEC OSF/1 [9] and MkLinux [10]. Further, QNX [32] is a widely adopted microkernel-based multiprocessor operating system for real-time tasks. However, although the microkernel implements lower-level handling in the system, a ported monolithic kernel still needs to provide locks around critical areas of the system.

An alternative approach is used in multiserver operating systems [6, 35]. Multiserver systems organize the system as multiple separated servers on a microkernel. These servers rely on microkernel abstractions such as threads and address spaces, which can in principle be backed by multiple processors transparently to the operating system servers. However, adapting an existing kernel to run as a multiserver system [13, 33] requires major refactoring of the kernel. Designing a system from scratch is a major undertaking, so in most cases it is more feasible to port an existing kernel.

2.3 Asymmetric Operating Systems

Like systems which use coarse-grained locking, master-slave systems (refer to Chapter 9 in [36]) 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 (“slaves”) run user-level applications. On system calls and other operations involving the kernel, master-slave systems divert the execution to the master processor. Commonly, this is done through splitting the ready queue into one slave queue and one master queue. Processes are then enqueued in the master queue on kernel operations, and enqueued in the slave queue again when the kernel operation finishes. Since all kernel access is handled by one processor, this method limits throughput for kernel-bound applications.

The master-slave approach is rarely used in current multiprocessor operating systems, but was more common in early multiprocessor implementations. For example, Goble and Marsh [14] describe an early tightly coupled VAX multiprocessor system, which was organized as a master-slave system. The dual VAX system does not split the ready queue, but instead lets the slave processor scan the ready queue for processes not executing kernel code. Also, although both processors can be interrupted, all interrupt handling (except timer interrupts) are done on the master processor. Our approach is a modern refinement of the master-slave approach, where the source code of the original system (“master”) remains unchanged.

An interesting variation of multiprocessor kernels was presented in Steven Muir’s PhD. thesis [31]. Piglet [31] 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 also 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 off-loaded to the LDK, and the Linux kernel and user-space processes communicate through lock-free message buffers with the LDK. A similar approach has also been used to offload the TCP/IP stack recently [34]. These approaches are beneficial if I/O-handling dominates the OS workload, whereas it is a disadvantage in systems with much computational work when the processors would serve better as computational processors. It can also require substantial modification of the original kernel, including a full multiprocessor adaption when more than one processor is running applications.

2.4 Cluster-based Approaches

Several approaches based on virtualized clusters have also been presented. One example is the Adeos Nanokernel [40] where a multiprocessor acts as a cluster with each processor running a modified version of the Linux kernel. The kernels cooperate in a virtual high-speed and low-latency network. The Linux kernel in turn runs on top of a bare-bones kernel (the Adeos nanokernel) and most features of Linux have been kept unchanged, including scheduling, virtual memory, etc. This approach has also been used in Xen [5], which virtualizes Linux or NetBSD systems.

Another cluster-based method is Cellular Disco [15], where virtualization is used to partition a large NUMA multiprocessor into a virtual cluster which also provides fault-containment between the virtualized operating systems. The virtualized systems provide characteristics similar to our approach in that they avoid the complexity issues associated with a traditional parallelization approach. However, they also require a different programming model than single-computer systems for parallel applications. Cluster-based approaches are also best suited for large-scale systems where scalability and fault tolerance are hard to achieve using traditional approaches.

MOSIX [4] is a single system image distributed system which redirects system calls to the “unique home node” of the process, thereby utilizing the central idea behind master-slave systems. MOSIX can distribute unmodified Linux applications throughout a cluster of asymmetric hardware. MOSIX is similar to our approach in that it redirects system calls, but has a different goal (providing a single-system image distributed system).

3 The Application Kernel Approach

All of the approaches presented in the last section require, to various degrees, extensive knowledge and modifications of the original kernel. We therefore suggest a different approach, the *application kernel approach*, which allows adding multiprocessor support with minimal effort and only basic knowledge about the original kernel. In this section we describe the general ideas behind the application kernel approach and an overview of how it works.

3.1 Terminology and Assumptions

Throughout the paper, we assume that the implementation platform is the Intel IA-32 although the approach is applicable to other architectures as well. We will follow the Intel terminology when describing processors, i.e., the processor booting the computer will be called the *bootstrap processor* while the other processors in the system are called *application processors*.

Also, we use a similar naming scheme for the two kernels: the original uniprocessor kernel is called the *bootstrap kernel*, i.e., the Linux kernel in the implementation described in this paper, whereas the second kernel is called the *application kernel*. Further, in order to not complicate the presentation, we will assume single-threaded processes in the discussion, although multi-threaded processes are also supported using the same technique.

3.2 Overview

The basic idea in our approach is to run the original uniprocessor kernel *as it is* on the bootstrap processor while all other processors run the application kernel. Applications execute on both kernels, with the application kernel handling the user-level part and the bootstrap kernel handling kernel-level operations. One way of describing the overall approach is that the part of the application that needs to communicate with the kernel is executed on a single bootstrap processor while the user-level part of the program is distributed among the other processors in the system, i.e., similar to master-slave kernels.

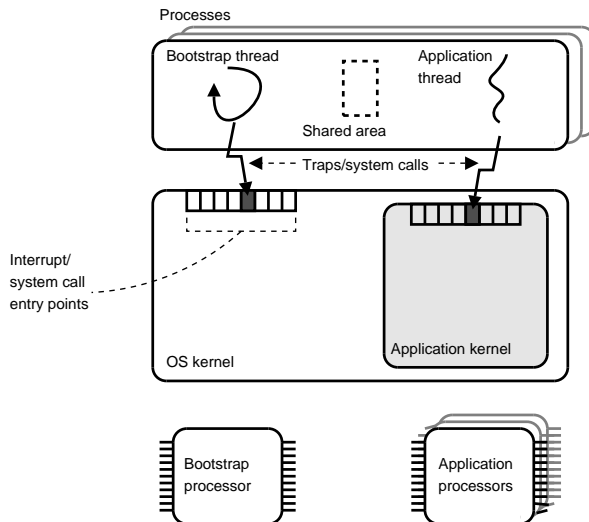


Figure 1: Overview of the application kernel approach.

Figure 1 shows an overview of the application kernel approach. The upper boxes represent user processes and the lower shows the bootstrap kernel and the application kernel. Each process has two threads, a *bootstrap thread* and an *application thread*. The bootstrap thread executes on the

bootstrap kernel, i.e., Linux, while the application threads are handled by the application kernel. An application thread runs the actual program code whereas the bootstrap thread serves as a proxy forwarding kernel calls to the bootstrap kernel. Note that the application kernel and the bootstrap kernel use unique interrupt and trap handlers to enable the application kernel to catch traps and faults caused by the application.

The two threads in the process' communicate through a shared area in the process address space. The bootstrap monitors the shared area to detect new system calls etc. Applications run as before, except when performing operations involving the kernel. On such events, the application thread traps into the application kernel, which then enters a message in the communication area. The actual event will be handled at a later stage by the bootstrap thread, which performs the corresponding operation. We will describe trap handling in detail in Section 3.4.

With the application kernel approach, we can add multiprocessor support to an existing operating system without neither doing modifications to the original operating system kernel, nor do we have to do any changes to the applications (not even recompiling them). There are a few special cases that might require kernel source changes, but those were not needed for our research prototype. Section 4.1 describes these special cases.

Compared to the other porting methods, our approach tries to minimize the effort needed to implement a multiprocessor port of a uniprocessor operating system. The focus is therefore different from traditional porting methods. Master-slave kernels, which are arguably most similar to our approach, place most of the additional complexity in the original kernel whereas we put it into two separate entities (the application kernel and the bootstrap thread). In a sense, our approach can be seen as a more general revitalization of the master-slave idea. The Cache Kernel [7, 16] employs a scheme similar to ours on redirecting system calls and page faults, but requires a complete reimplementaion of the original kernel to adapt it to the cache kernel. We can also compare it to the MOSIX system [4] which also redirects system calls, although MOSIX is used in a cluster context and has different goals than the application kernel.

3.3 Hardware and Software Requirements

The application kernel approach places some restrictions (often easy to fulfill) on the processor architecture and the bootstrap kernel. The architecture must support at least the following:

1. Binding of external interrupts to a specific processor and at the same time allow CPU-local timer interrupts.
2. Retrieving the physical page table address of the currently running process.
3. Interrupt and trap handlers must be CPU-local.

The first requirement must be fulfilled since only the bootstrap kernel handles all external interrupts except for timer interrupts. Timer interrupts need to be CPU-local for scheduling

to take place on the application kernel. On the IA-32 this is possible to implement with the APIC (Advanced Programmable Interrupt Controller), which has a per-processor timer. MIPS uses a timer in the coprocessor 0 on the processor chip [30] and PowerPC has a decremter register [19] which can be used to issue interrupts. The interrupt *handlers* must be private for different processors, which is directly possible on IA-32 processors through the Interrupt Descriptor Table, IDT. For architectures where the interrupt handlers reside on fixed addresses, e.g., MIPS, instrumentation of the interrupt handlers are needed.

Our approach also places two requirements on the bootstrap kernel. First, it must be possible to extend the kernel with code running in supervisor mode. This requirement is satisfied in most operating systems, e.g., through loadable modules in Linux. Second, the bootstrap kernel must not change or remove any page mappings from the application kernel. The application kernel memory needs to be mapped to physical memory at all times, since revoking a page and handing it out to a process (or another location in the kernel) would cause the application kernel to overwrite data for the bootstrap kernel or processes.

3.4 Application Kernel Interaction

Figure 2 shows how the kernel interaction works in the application kernel approach. Kernel interaction requires 8 steps, which are illustrated in the Figure. In the discussion, we assume that the operation is a system call, although page faults and other operations are handled in the same way.

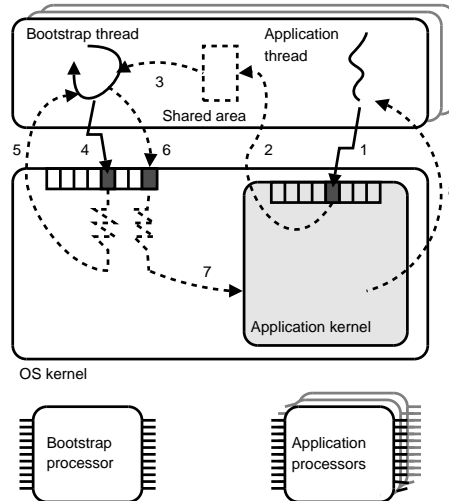


Figure 2: System call/trap handling in the application kernel approach

1. The application (i.e., the application thread running on one of the application processors) issues a system call and traps down to the application kernel. This is handled by the CPU-

local trap vector.

2. The application kernel enters information about the call into the shared area, and thereafter schedules another thread for execution.
3. At a later point, the bootstrap thread wakes up and finds a message in the shared area.
4. The bootstrap thread then parses the message and performs the corresponding operation (i.e., issuing the same system call in this case).
5. The bootstrap kernel will thereafter handle the system call from the bootstrap thread and return control to the bootstrap thread.
6. After this, the bootstrap thread must tell the application kernel that the application thread can be scheduled again. Since the application kernel runs as a loadable module within the bootstrap kernel, it must do this through the driver interface of the bootstrap kernel, issuing the application kernel **apkern_activate_thread** call.
7. The application kernel driver, running on the bootstrap processor, enters the application thread into the ready queue again.
8. Finally, the application thread is scheduled at a later point in time on one of the application processors.

The `clone` and `fork` system calls are handled slightly different than other calls, and are described in detail in Section 4.2. Further, the `exit` system call and exceptions that cause process termination (for example illegal instructions) are different than page faults and other system calls. This is because the bootstrap kernel is unaware of the application thread and will terminate the process without notifying the application kernel. If this is not handled, the application kernel will later schedule a thread which runs in a non-existing address space. For this case, step 2 of the algorithm above is modified to clean up the application thread (i.e., free the memory used by the thread control block and remove the thread from any lists or queues).

Another special case is when the information flows the opposite way, i.e., when the kernel asynchronously activates a process (for instance in response to a signal in Linux). In this case, the handler in the bootstrap thread will issue the **apkern_activate_thread** call directly, passing information about the operation through the shared area. The application kernel will then issue the same signal to the application thread, activating it asynchronously. Our current implementation does not support asynchronous notifications, but it would be achieved by registering signal handlers during the bootstrap thread startup phase.

3.5 Exported Application Programming Interface

The application kernel API is only available via driver calls to the bootstrap kernel. There is no way to call the application kernel directly via system calls in the application thread since the trap handling matches that of the bootstrap kernel and only forwards the events through the shared

area. A straightforward way of allowing direct calls to the application kernel would be to use a different trap vector than the Linux standard, which could be used e.g., to control application kernel scheduling from applications. The exported interface consists of six calls:

- **apkern_init**: This routine is called once on system startup, typically when the application kernel device driver is loaded. It performs the following tasks:
 - It initializes data structures in the application kernel, for example the ready-queue structure and the thread lookup table.
 - It starts the application processors in the system. On startup, each processor will initialize the interrupt vector to support system calls and exceptions. The processor will also enable paging and enter the idle thread waiting for timer interrupts.
- **apkern_thread_create**: This function is called from the bootstrap thread when the process is started. The function creates a new thread on the application kernel. The thread does not enter the ready queue until the **apkern_thread_start** call is invoked.
- **apkern_thread_ex_regs**: Sometimes it is necessary to update the register contents of a thread (for example copying the register contents from parent to child when forking a process), and the application kernel therefore has a call to “exchange” the register contents of a thread.
- **apkern_thread_get_regs**: This function returns in the current register context of a thread (also used with fork).
- **apkern_thread_start**: Place a thread in the ready queue.
- **apkern_thread_activate**: Thread activation is performed when the bootstrap thread returns, e.g., from a system call, to wake up the application thread again. The call will enter the application thread back into the ready queue and change its state from *blocked* to *ready*.

4 Implementation

We implemented the application kernel as a loadable kernel module for Linux. The module can be loaded at any time, i.e., the application kernel does not need to be started during boot but can be added when it is needed. Since modules can be loaded on demand, the application kernel can also be started when the first process uses it. It is further not necessary to recompile applications to run on the application kernel, and applications running on the application kernel can coexist seamlessly with applications running only on the bootstrap processor.

The layout of the shared memory area for the Linux implementation is shown in Figure 3. The shared area data type, **apkern_comm_entry_t** has a union with the different types of messages, with page faults and system calls shown in the figure and a variable (**bootstrap_has_msg**) which is used by the application kernel to signal to the bootstrap thread. There is always a one to one mapping between application threads and bootstrap threads, i.e., multithreaded processes

will have several bootstrap thread. The bootstrap thread does not respond to system calls etc., through the shared area, but invokes the application kernel driver instead. Since the shared area is a one-way communication channel, it needs no explicit protection.

```
typedef struct {
    volatile bool_t bootstrap_has_msg;
    volatile apkern_comm_nr_t nr;
    volatile addr_t pc;

    union {
        struct {
            volatile addr_t addr;
            volatile bool_t write;
        } PACKED pagefault;

        struct {
            volatile uint_t nr;
            volatile uint_t arg1;
            ...
            volatile uint_t arg6;
            volatile uint_t ret;
        } PACKED syscall;
        ...
    } u;
} apkern_comm_entry_t;
```

Figure 3: Shared area layout

The application kernel is initialized, i.e., processors are booted etc., when the kernel module is loaded. The application kernel is thereafter accessible through a normal Linux device file, and a process that wants to run on the application kernel opens the device file on startup and closes it when it exits (this can be done automatically and is described in Section 4.3). All interactions with the application kernel, apart from `open` and `close`, are done using `ioctl` calls, through which the exported interface is available.

Figure 4 illustrates the application kernel driver (a char-type device) structure and an `apkern_activate_thread` call. The call from the bootstrap thread enters through the Linux system call handler which forwards it to the `ioctl` entry point for the device driver. The `ioctl` handler in turn updates the thread control block for the activated thread, locks the application kernel ready queue, and enters the thread control block into the ready queue. In the rest of this section, we will discuss details related to paging, forking and application startup from the Linux implementation of the application kernel.

4.1 Paging

All page faults are handled in the bootstrap thread by setting the stack pointer to the address of the page fault and touching that memory area. Although this seems like an unnecessary step

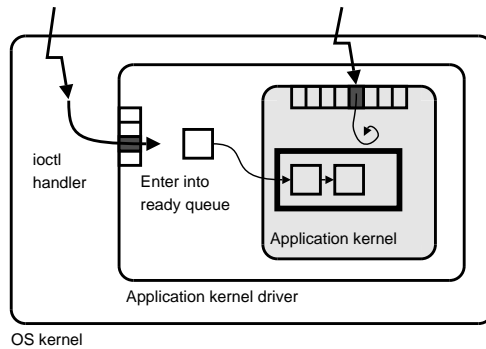


Figure 4: Application kernel device driver structure

instead of just accessing the memory directly, it is needed as a workaround since Linux terminates the program if stack access is done below the current stack pointer.

The paging implementation also illustrates the one case where the application kernel approach might require kernel modifications. The problem (which is general and affects other approaches as well) occurs in multi-threaded processes on page table updates, when the TLB contents for different processors running in the same address space will be inconsistent¹. For example, if processor 0 and 1 execute threads in the same address space, and processor 0 revokes a page mapping, the TLB of processor 1 will contain an incorrect cached translation. To solve this, an inter-processor interrupt is invoked to invalidate the TLB of the other processors, which requires changes to the page fault handling code. In our prototype, we ran without disk swap and the IPIs are therefore not needed and have not been implemented.

4.2 clone/fork System Calls

The Linux `clone` and `fork` system calls require special handling in the application kernel. Both calls start a new process which inherits the context of the invoking thread. The difference is that `clone` allows for sharing the address space with the parent (creating a new thread), while `fork` always separate the address spaces (creating a new process). `clone` also requires the invoker to specify a callback function that will be executed by the cloned thread. In Linux, `fork` is simply a special case of `clone`, although the implementation of `fork` predates `clone`.

We illustrate the steps needed in a `clone` or `fork` call in Figure 5. If we would just issue the system call directly, the bootstrap thread would run the cloned thread itself. Therefore we first clone the bootstrap thread, then let the cloned bootstrap thread create a new application kernel thread (i.e., handling the original clone), and finally enters the bootstrap thread loop. This effectively splits the `clone` call in two, creating a new thread pair. The `fork` call works the same way, but has different return semantics, i.e., it returns “twice” instead of using a callback.

¹On architectures with tagged TLBs, e.g., MIPS, this could occur even in single-threaded processes since the TLB is not necessarily flushed on page table switches.

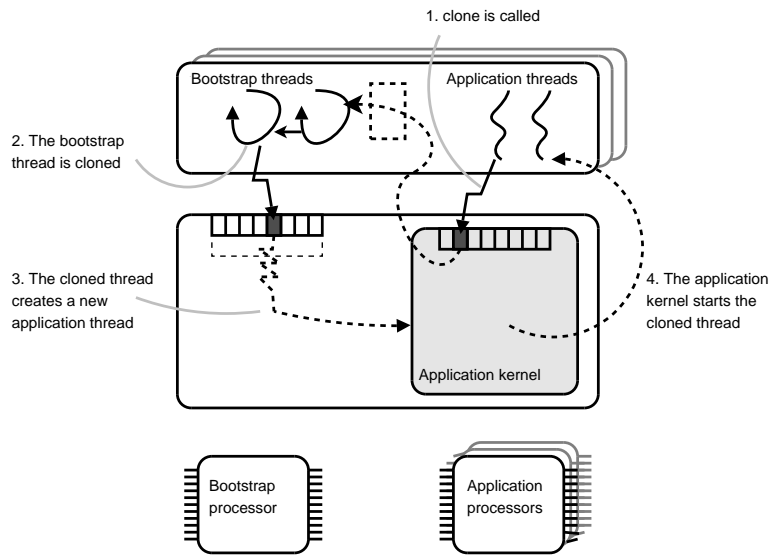


Figure 5: Handling of the `clone` system call

4.3 Running Applications

Our implementation allows running dynamically linked applications directly, without modifying or even recompiling them. It is also possible to run a mixed system, with some applications running on the application kernel whereas others are tied to the bootstrap processor.

We achieve this by applying some knowledge about application startup under Linux. In Linux, applications are started by a short assembly stub which in turn calls `__libc_start_main`. This function, provided by GNU libc, starts the `main` function. The `__libc_start_main` function is dynamically linked into the executable and can therefore be overridden. We override `__libc_start_main` with the startup routine for the application kernel, which can be done as long as the application is dynamically linked against libc. To run a process on the application kernel, we simply set the `LD_PRELOAD` environment variable to preload a library with the bootstrap thread implementation.

The overriding process is illustrated in Figure 6. The overridden `__libc_start_main` will just invoke the original `__libc_start_main`, but with `apkern_thread` instead of `main` as starting function. This function in turn will either, depending on if the `NOAPPKERN` environment variable is set, invoke the original `main` and thereby bypassing the application kernel, or start the bootstrap thread.

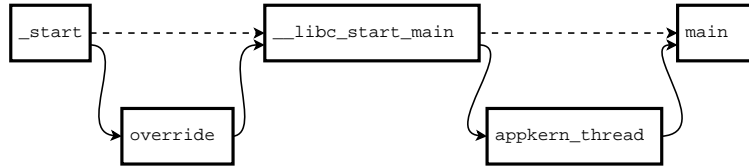


Figure 6: Application startup. The dashed lines shows the original execution while the solid lines show the overridden execution path.

5 Experimental Setup and Methodology

We have conducted an evaluation of the application kernel approach where we evaluate both latency and throughput. First, we measure single-process performance in order to estimate the extra latency caused by the application kernel. Second, we measure scalability of multiprogramming and parallel benchmarks. In the evaluation, we use standard UNIX tools, the SPLASH 2 [?] benchmarks and the SPEC CPU2000 [38] benchmark suite. Further, we have also evaluated the implementation size and complexity of our approach, which was performed by counting the physical lines of code in the application kernel and calculating the McCabe cyclomatic complexity [12] which gives the number of independent code paths through a function. The code lines were counted with the sloccount tool [39] by David A. Wheeler and the cyclomatic complexity was measured by the pmccabe tool by Paul Bame [3].

5.1 Evaluation Environment

We performed our performance evaluation using the Simics full system simulator [28] and real hardware. We setup Simics to simulate a complete IA-32 system with 1 to 8 processors. Our hardware is a 200MHz dual Pentium Pro with 8KB first-level instruction and data caches, and a 256KB per-processor L2 cache. The Simics simulator allows us to use unmodified hard disk images, containing the complete operating system. Compared to real hardware, our simulated setup does not simulate caches in the system, and some other performance issues relevant in multiprocessor systems [?], such as costs associated with data alignment, cross-processor cache access etc., are not accounted for in our simulations. Our prototype also has known performance issues, e.g., we have not optimized the memory layout for efficient use of the cache. However, the fundamental limitation of the application kernel approach is that the bootstrap thread at some point will be a scalability bottleneck. We believe that the simulated measurements give a good indication of when this bottleneck is reached for various usage patterns.

The execution on hardware serves to validate the correctness of our implementation in a real

setting, and is also used to establish the latency for kernel operations with the application kernel. We successfully ran all the benchmarks on our hardware as well as on the simulated system.

We benchmarked uniprocessor Linux with the application kernel module against multiprocessor Linux, running the 2.4.26 version of the kernel, henceforth referred to as SMP Linux. Our experiments report the time required to execute the benchmarks in terms of clock cycles on the bootstrap processor. Our system was a minimal Debian GNU/Linux 3.1 (“Sarge”)-based distribution, which ran nothing but the benchmark applications.

Table 1: The benchmarks used in the performance evaluation

Benchmark	Command	Description
Single-process benchmarks		
find	<code>find /</code>	List all files in the system (13946 files and directories)
SPEC2000 gzip	<code>164.gzip lgred.log</code>	Compression of a logfile, computationally intensive.
SPEC2000 gcc	<code>176.gcc smred.c-iterate.i -o a.s</code>	SPEC 2000 C-compiler.
Parallel benchmarks		
SPLASH2 RADIX	<code>RADIX -n 8000000 -p8</code>	Sort an array with radix sort, 8 threads.
SPLASH2 FFT	<code>FFT -m20 -p8</code>	Fourier transform, 8 threads.
SPLASH2 LU (non-contiguous)	<code>LU -p 8 -b 16 -n 512</code>	Matrix factorization, 8 threads.
Multiprogramming benchmarks		
176.gcc	<code>176.gcc smred.c-iterate.i -o a.s</code>	SPEC2000 C-compiler
find	<code>find /</code>	List all files in the system (13946 files and directories).
grep	<code>grep "linux" /boot/System.map</code>	Search for an expression in a file. System.map has 150,000 lines.
find and grep	<code>find / grep "data"</code>	List all files in the system and search for a string in the results.
SPLASH2 FFT	<code>FFT -m10 -p8</code>	Fourier transform, 8 threads.
SPLASH2 LU	<code>LU -p 8 -b 16 -n 512</code>	Matrix factorization, 8 threads.

5.2 Benchmarks

For the performance evaluation, we conducted three types of performance measurements. First, we ran a number of single-process benchmarks to evaluate the overhead caused by the system call forwarding used by the application kernel approach. These benchmarks run one single-threaded process at a time and should therefore be unaffected by the number of processors. Second, we also ran a set of multithreaded parallel applications, which shows the scalability of compute-bound applications. Third, we also evaluated a multiprogramming workload. In the multiprogramming

benchmark, we ran a set of programs concurrently and measured the duration until the last program finished. This benchmark should be characteristic of a loaded multi-user system.

The programs we used are a subset of the SPEC CPU2000 benchmarks, a subset of the Stanford SPLASH 2 benchmarks, and a set of standard UNIX tools. For SPEC CPU2000, we used the Minnespec reduced workloads [22] to provide reasonable execution times in our simulated environment. The SPLASH 2 benchmarks were compiled with a macro package which uses `clone` for the threading implementation and `pthread` primitives for mutual exclusion. The SPLASH SPEC benchmarks were compiled with GCC version 3.3.4 (with optimization `-O2`) and the UNIX applications were unmodified Debian binaries. The benchmark applications are summarized in Table 1.

6 Experimental Results

In this Section, we describe the results obtained from our measurements. Table 3 and 4 show the speedup vs. uniprocessor Linux for SMP Linux and the application kernel. For the parallel and multiprogramming benchmarks, the speedup is also shown in Figure 7. The results from the `getpid` evaluation is shown in Table 2.

6.1 Performance Evaluation

On our hardware, issuing a `getpid` call takes around 970 cycles in Linux on average (the value fluctuates between 850 and 1100 cycles) whereas the same call requires around 5700 cycles with the application kernel as shown in Table 2. In Simics, the cost of performing a `getpid` call is 74 cycles in Linux and around 860 cycles with the application kernel. Since `getpid` performs very little in-kernel work, the cost for Linux is dominated by the two privilege level switches (user mode to kernel and back). For the application kernel, there are five privilege level switches (see Figure 2). First, the application thread traps down into the application kernel, which updates the shared area. The bootstrap thread thereafter performs another trap for the actual call and upon return invokes the application kernel driver through an `ioctl` call, i.e., performing another three privilege level switches. Finally, the application thread is scheduled again, performing the fifth privilege level switch. In our simulated system, each instruction executes in one cycle and there is no additional penalty for changing privilege mode and therefore the `getpid` cost is dominated by the number of executed instructions. This explains why the application kernel overhead is proportionally larger in the simulated system than on real hardware.

Table 2: `getpid` latency in Linux and the application kernel

	Linux	Application Kernel
PPro 200MHz	970	5700
Simics	74	860

Table 3: Speedup for the single-process benchmarks.

Processors	Speedup vs. uniprocessor Linux					
	Find		176.gcc		164.gzip	
	Linux	Appkern	Linux	Appkern	Linux	Appkern
2	0.9803	0.7844	1.0015	0.8976	1.0008	0.9461
3	0.9795	0.8284	1.0033	0.9125	1.0012	0.9461
4	0.9807	0.8641	1.0047	0.9218	1.0014	0.9462
5	0.9804	0.8690	1.0053	0.9230	1.0016	0.9462
6	0.9800	0.8748	1.0047	0.9244	1.0016	0.9462
7	0.9795	0.8784	1.0050	0.9252	1.0017	0.9462
8	0.9776	0.8831	1.0055	0.9260	1.0017	0.9462

Table 4: Speedup for the parallel and multiprogramming benchmarks.

Processors	Speedup vs uniprocessor Linux							
	RADIX		FFT		LU		Multiprogramming	
	Linux	Appkern	Linux	Appkern	Linux	Appkern	Linux	Appkern
2	2.0433	1.0834	1.6916	1.0401	1.9217	1.2662	1.5049	0.9705
3	3.3758	2.5174	2.2930	1.8654	2.9430	2.0795	1.6627	1.1375
4	4.0885	3.7227	2.5090	2.3235	3.5053	2.9941	1.6850	1.1779
5	5.1898	4.8200	2.8456	2.6323	4.0857	3.8009	1.6782	1.1878
6	5.9562	5.5736	2.9927	2.8626	4.7706	5.0445	1.6845	1.1962
7	6.9355	6.1934	3.1732	3.0188	5.3277	5.1628	1.6803	1.2059
8	8.0009	6.0924	3.3272	3.0745	6.0084		1.6839	

In the computationally intensive single-process `gcc` and `gzip` benchmarks from SPEC CPU2000, the application kernel performs almost on-par with SMP Linux (the difference is between 5 and 10%) as shown in Table 3. Further, we can also see that as more processors are added, the gap decreases because there is a higher probability of a processor being free to schedule the thread when the bootstrap thread has handled the call.

A weak spot for the application kernel shows in the filesystem-intensive `find` benchmark. Here, the overhead associated with forwarding system calls prohibit the application kernel to reach SMP Linux performance levels. However, since application kernel applications can coexist seamlessly with applications tied to the bootstrap kernel, it is easy to schedule these applications on the bootstrap kernel.

The selected computationally intensive parallel benchmarks from the Stanford SPLASH 2 suite exhibit good scalability both in SMP Linux and for the application kernel (see Table 4 and Figure 7). The results for the application kernel are close to those for SMP Linux, especially considering that the application kernel excludes one of the processors (the bootstrap processor) for computation. This shows that the application kernel is a feasible approach for computationally intensive applications, where the kernel interaction is limited.

The multiprogramming benchmark, also shown in Table 4 and Figure 7, contains a mix of applications which have different behavior in terms of user/kernel execution. For this benchmark, we see that running all applications on the application kernel places a high strain on the bootstrap

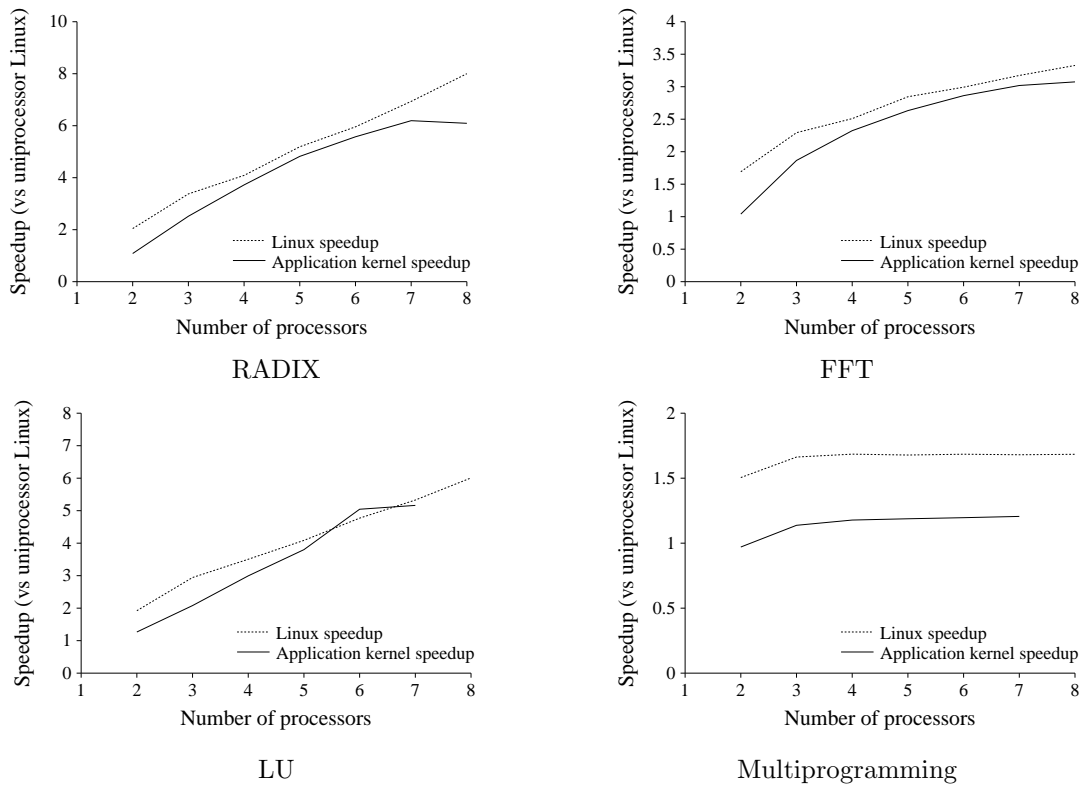


Figure 7: Speedup for the parallel and multiprogramming benchmarks vs. uniprocessor Linux.

kernel, which hampers the scalability compared to SMP Linux. For general multiprogramming situations, it is probably better to divide the processes so that kernel-bound processes run on the bootstrap processor while the rest are executed on the application kernel.

6.2 Implementation Complexity and Size

The application kernel was ported from the implementation presented in [23], and most of the internals of the kernel are completely unchanged. Apart from some restructuring and the loadable Linux kernel module, the only changes to the actual application kernel is some low-level handling of system calls (i.e., the used trap vector and parameter passing). One single developer spent seven weeks part-time implementing the application kernel support for Linux. The previous implementation took about five weeks to finish, and was also done by a single developer.

The number of physical code lines (not counting empty and comments) in the application kernel is 3,600. Of these, the Linux driver module takes up around 250 lines, roughly equally split in initialization and handling of `ioctl` calls. Only around 400 lines of the implementation were changed from our previous implementation. Libraries, a small part of `libc` and `malloc`, `list`, `stack` and `hash table` implementations, account for another 920 lines of code. The user-level library which contains the bootstrap thread consists of 260 lines of code. Roughly one third of these are

needed for the handling of `clone` and `fork` while around 70 lines are needed for startup. The rest is used in the implementation of page fault and system call handling (excluding `clone` and `fork`). The code lines are summarized in Table 5.

Table 5: Comment-free lines of code

Category	Lines of code
Application kernel	2,400
Linux driver	360
Libraries	920
Bootstrap thread	260

The source consists of around 360 lines of assembly code and the rest being C-code. The high proportion of assembly code, almost 10%, stems from the fact that a fairly large part of the code deals with startup of the application processors and low-level interrupt handling. If we disregard the library code (which is independent of the application kernel), the assembly portion increases to 17%.

A histogram of the McCabe cyclomatic complexity for the application kernel (without the library implementation), and the kernel core and the IA-32-specific parts of Linux 2.4.26, FreeBSD 5.4 and L4/Pistachio 0.4 [?] is shown in Figure 8. As the figure indicates, the cyclomatic complexity of the application kernel implementation is fairly low (a value below 10 is generally regarded as indicative of simple functions). Compared to the other kernels, we can see that the application kernel has a larger proportion of functions with low cyclomatic complexity than especially Linux and FreeBSD.

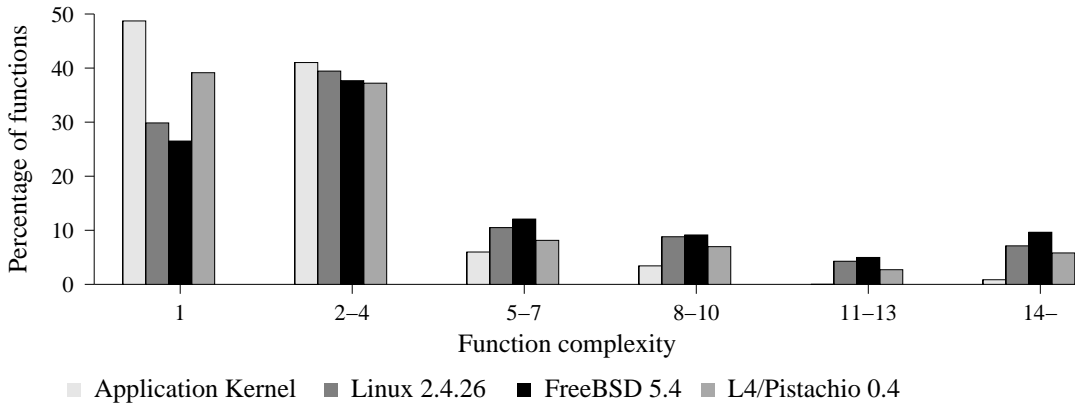


Figure 8: Histogram of McCabe cyclomatic complexity for the Application Kernel, Linux 2.4.26, FreeBSD 5.4 and the L4/Pistachio 0.4 microkernel.

7 Conclusions and Future Work

In this paper we have presented the application kernel, an alternative approach for adding SMP support to a uniprocessor operating system. Our approach has lower implementation complexity than traditional approaches, often without changes to the original uniprocessor kernel, while at the same time providing scalable performance. In this sense, the application kernel approach can be seen as a modern revitalization of the master-slave approach. There are also similarities with approaches used in distributed systems.

We have evaluated a prototype implementation of the application kernel approach for a uniprocessor Linux kernel, where the results show that our approach is a viable method to achieve good performance in computationally intensive applications. We also show that the implementation is quite straightforward, with a low cyclomatic complexity compared to other operating system kernels and a small size (around 3,600 lines) requiring only seven weeks to implement.

There are several advantages with our approach. First, we do not need to modify the large and complex code of the uniprocessor kernel. Second, the development of the uniprocessor kernel can continue as usual with improvements propagating automatically to the multiprocessor version. Our evaluation also shows that a large portion of the effort of writing the application kernel can be reused for other uniprocessor kernels which leads us to believe that our approach and implementation is fairly generic and reusable for other kernels.

There are a number of optimizations possible for the application kernel approach. For instance, some threads could run entirely on the bootstrap kernel, which would mainly be interesting for kernel-bound applications. A migration scheme similar to that in MOSIX could then be used to move kernel-bound threads to the bootstrap processor during runtime. Further, some system calls should be possible to implement directly on the application kernel, providing the semantics of the system calls are known. For example, sleeping, yielding the CPU and returning the process ID of the current process can easily be implemented in the application kernel.

Acknowledgments and availability

We would like to thank the anonymous reviewers for their useful feedback. 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>).

The application kernel source code is available as free software licensed under the GNU GPL at http://www.ipd.bth.se/ska/application_kernel.html.

References

- [1] A. C. Arpaci-Dusseau and R.H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 43–56, 2001.
- [2] M. J. Bach and S. J. Buroff. Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical Journal*, 63(8):1733–1749, October 1984.
- [3] Paul Bame. pmccabe. See <http://parisc-linux.org/~bame/pmccabe/>, accessed 20/6-2004.
- [4] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, pages 361–372, March 1999.
- [5] Paul T. Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating systems principles*, pages 164–177, 2003.
- [6] M. I. Bushnell. The HURD: Towards a new strategy of OS design. *GNU's Bulletin*, 1994. Free Software Foundation, <http://www.gnu.org/software/hurd/hurd.html>.
- [7] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 179–193. USENIX Association, November 1994.
- [8] R. Clark, J. O'Quin, and T. Weaver. Symmetric multiprocessing for the AIX operating system. In *Compcon '95. 'Technologies for the Information Superhighway', Digest of Papers.*, pages 110–115, 1995.
- [9] J. M. Denham, P. Long, and J. A. Woodward. DEC OSF/1 symmetric multiprocessing. *Digital Technical Journal*, 6(3), 1994.
- [10] F.B. des Places, N. Stephen, and F.D. Reynolds. Linux on the OSF Mach3 microkernel. In *Proceedings of the Conference on Freely Distributable Software*, February 1996.
- [11] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.
- [12] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.
- [13] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *9th ACM/SIGOPS European Workshop "Beyond the PC: Challenges for the operating system"*, pages 109–114, Kolding, Denmark, September 2000.
- [14] George H. Goble and Michael H. Marsh. A dual processor VAX 11/780. In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, pages 291–298, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

- [15] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the ACM Symposium on Operating Systems Principle (SOSP'99)*, pages 154–169, Kiawah Island Resort, SC, December 1999.
- [16] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 123–136, Berkeley, October 28–31 1996. USENIX Association.
- [17] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.
- [18] H. Härtig, M Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of u-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, St. Malo, France, October 1997.
- [19] IBM Corporation. *PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors*, March 2005.
- [20] J. Kahle. Power4: A dual-CPU processor chip. In *Proceedings of the 1999 International Microprocessor*, San Jose, CA, October 1999.
- [21] 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.
- [22] A. KleinOowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture News Letters*, 1, June 2002.
- [23] 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.
- [24] 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.
- [25] G. Lehey. Improving the FreeBSD SMP implementation. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 155–164. USENIX, June 2001.
- [26] J. Liedtke. On u-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 1–14, Copper Mountain Resort, CO, December 1995.
- [27] Robert Love. *Linux Kernel Development*. Sams, Indianapolis, Indiana, 1st edition, 2003.

- [28] 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.
- [29] 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.
- [30] MIPS Technologies. *MIPS32 Architecture for Programmers Volume III: The MIPS32 Privileged Resource Architecture*, March 2001.
- [31] Stephen James Muir. *Piglet: an operating system for network appliances*. PhD thesis, University of Pennsylvania, 2001.
- [32] QNX Software Systems Ltd. *The QNX Neutrino Microkernel*, see <http://qdn.qnx.com/developers/docs/index.html>, 2005. Accessed 28/7-2005.
- [33] Freeman L. Rawson III. Experience with the development of a microkernel-based, multi-server operating system. In *6th Workshop on Hot Topics in Operating Systems*, pages 2–7, Cape Cod, Massachusetts, USA, May 1997.
- [34] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, 2004.
- [35] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, Queens’ College, University of Cambridge, April 1995.
- [36] C. Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, Boston, 1st edition, 1994.
- [37] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA-11)*, pages 248–252, San Francisco, CA, USA, February 2005.
- [38] Standard Performance Evaluation Corporation. *SPEC CPU 2000*, see <http://www.spec.org>, 2000.
- [39] David A. Wheeler. Sloccount, see <http://www.dwheeler.com/sloccount/>, Accessed 20/6-2005.
- [40] K. Yaghmour. *A practical Approach to Linux Clusters on SMP hardware*, July 2002. See <http://www.opersys.com/publications.html>, accessed 28/7-2005.
- [41] R. Zhang, T. F. Abdelzaher, and J. A. Stankovic. Kernel support for open QoS computing. In *Proceedings of the 9th IEEE Real-Time/Embedded Technology And Applications Symposium (RTAS)*, pages 96–105, 2003.