A novel method for adding multiprocessor support to a large and complex uniprocessor kernel

Simon Kågström, Lars Lundberg, Håkan Grahn
School of Engineering,
Blekinge Institute of Technology
S-372 25 Ronneby, Sweden

E-mail: {ska, llu, hgr}@bth.se

Abstract

The current trend of using multiprocessor computers for server applications requires operating system adoptions for high performance. 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 kernel as a "black box", to which no or 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.*

A prototype implementation illustrates that the approach can be realized with fairly small resources. We also present an initial performance evaluation where we show that the performance is good for user-bound applications.

1. Introduction

Uniprocessor computers are now being replaced with (small) multiprocessors for performance reasons. Moreover, modern processor chips from Intel and other major manufacturers often contain more than one logical CPU core. For instance, current Intel Pentium 4 and Xeon processors contain two logical processors [18] and there are also other multiprocessor chips on the market and in research systems [13, 23, 3]. To take advantage of multiprocessing, good operating system support is crucial.

We are currently working on a project together with a major developer of industrial systems in Sweden. The company has since more than 10 years developed an operating system kernel for clusters of uniprocessor IA-32 hardware which has interesting properties such as fault tolerance and high performance (mainly in terms of throughput). In order to take advantage of the trend from uniprocessor Intel hardware to small shared-memory multiprocessors, a multiprocessor version of the kernel is being developed. The problems faced then were that it was extremely 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 led to a structure which is hard to grasp) and the intricate nature of OS kernels.

This situation 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, [1] proposes a method where "gray-box" knowledge about algorithms and behavior of an operating system is used to acquire control and information over the OS without explicit interfaces or OS modification. There has also been some work where the kernel is changed to provide quality of service guarantees to large unmodified applications [25].

For the kernel of our industrial partner, it turned out that the software engineering problems of adding multiprocessor support were extremely difficult 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 a somewhat unconventional approach. In our approach, we treat the existing kernel more or less as a black box and build the multiprocessor adoptions on top of the existing kernel. A custom kernel, of which the original kernel is unaware, is constructed to run beside the original kernel on the other processors. The original kernel continues to handle kernel access while unmodified applications are spread out over the other processors, redirecting system calls to the uniprocessor kernel. We expect this approach to substantially lower the development and maintenance costs compared to a traditional SMP

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

(Symmetric MultiProcessing) port.

In this paper, we describe the ideas behind this approach and the design decisions we made in an implementation of the approach for a small prototype uniprocessor kernel. Further, we present an evaluation of a prototype implementation where we show that the approach is feasible from both a complexity and performance viewpoint.

The paper is structured as follows: In Section 2, an overview of existing multiprocessor approaches is made, Section 3 presents our approach while Section 4 discusses design and implementation issues. Section 5 presents a performance evaluation and finally we conclude in Section 6.

2. Approaches for symmetric multiprocessing

There are several ways of structuring multiprocessor support for a kernel. In this section, we present the traditional approaches to SMP porting as well as some alternative methods.

2.1. Traditional SMP porting

Traditional SMP operating systems have often evolved from monolithic uniprocessor kernels. These uniprocessor kernels (for example Linux and BSD UNIX) contain large parts of the actual operating system, making SMP adaption 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. how large portions of code or data structures a lock protects, is very important for performance and maintainability reasons.

Early SMP operating systems often used coarse-grained locking. These systems employ a locking scheme where only one processor runs in the kernel (or in a kernel subsystem) at a time [22]. The main advantage with the coarse-grained method is that most data structures of the kernel can remain unprotected, and this simplifies the SMP implementation. In the most extreme case, a single “giant” lock protects the entire kernel.

Fine-grained locking allow several processors to execute in the kernel concurrently. Such systems allow for better scalability since processes can run without blocking on kernel-access but also require more careful implementation, since many data structures in the kernel must be locked. The FreeBSD SMP implementation, which originally used coarse-grained locking, have shifted toward a fine-grained method [15] and mature UNIX systems such as AIX and Solaris also implement SMP support with fine-grained locking [5, 14].

2.2. Microkernel-based systems

Another approach is to run the operating system on top of a microkernel. For example, L4Linux [11], a modified Linux kernel, runs on top of the L4 microkernel [16]. Also, the Mach microkernel has been used as the base for many operating systems, for example DEC OSF/1 [6] and MkLinux [7]. QNX [20] is also a widely adopted microkernel-based SMP operating system.

Microkernel-based systems potentially provides better system security by isolating operating system components (for example separating the network subsystem from device drivers etc) and also better portability since much of the hardware dependencies can be abstracted away by the microkernel. For example, multiserver operating systems [4, 21] provide a system structured from multiple separated servers which run on top of a microkernel. These servers rely on microkernel abstractions such as threads and address spaces, which could in principle be backed by multiple processors transparently to the operating system servers. However, adapting an existing kernel to run as a multiserver system (which has been attempted in [9]) requires major refactoring of the kernel.

2.3. Other approaches

Like systems which use coarse-grained locking, master-slave systems (refer to chapter 9 in [22]) 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. The master-slave approach is rarely used in current SMP operating systems.

An interesting variation of multiprocessor kernels was presented in [19]. The AsyMOS (Asymmetric Multiprocessor Operating System) divides the processors in a system between *application processors*, APs, and *device processors*, DPs. The APs run application code and most OS functionality whereas the DPs handle hardware devices. For example, a processor might be allocated to an Ethernet card to handle hardware interrupts and perform packet handling needed for the network subsystem. This approach is beneficial if I/O-handling dominates the OS workload, whereas it is a disadvantage in systems with much computational work (where the processors would serve better as computational processors). It also requires modification of the original kernel, including a full SMP adaption for more than one AP.

Several cluster-based approaches have also been presented. One example is [24] where an SMP system acts

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

as a cluster with each processor running a modified version of the Linux kernel. The kernels cooperate in a virtual high-speed, low-latency network. The Linux kernel in turn runs on top of a bare-bones kernel (the Adeos nanokernel) and most features of Linux has been kept intact, including scheduling, virtual memory etc. Another cluster-based method has been presented in [10], where virtualization is used to partition a large-scale multiprocessor into a virtual cluster. These systems provide characteristics similar to our approach in that they avoid the complexity issues associated with a traditional approach. However, the cluster-based systems require another programming model than single-computer systems. Cluster-based approaches are probably best suited for large-scale systems where scalability and fault tolerance is hard to achieve using traditional approaches.

3. The application kernel approach

Because of the intricate nature of our industrial partner's kernel, we wanted to avoid the problems associated with kernel complexity and size. All the approaches presented in the previous section require extensive knowledge about and access to the internals of the operating system kernel and we therefore suggest a radically different approach, the *application kernel approach*. In this section we describe the general ideas behind our approach and also discuss requirements on the hardware and the operating systems.

3.1. Overview of the approach

In the following discussion, 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 terminology for the two kernels: the original uniprocessor kernel is called the *bootstrap kernel* whereas the second kernel is called the *application kernel*. Further, we will assume single-threaded processes in the discussion below, although multi-threaded processes can also be supported with minor adaptations.

The basic idea in our approach is to run the original uniprocessor kernel *as it is* on the bootstrap processor. All other processors run the application kernel on which the user-part of the applications execute. One way of describing the overall approach is that the part of the application program that needs to communicate with the kernel (and hardware interrupts) is executed by the bootstrap processor and the other parts of the program are distributed among the other processors in the system, i.e. a mode of operation similar to master-slave kernels.

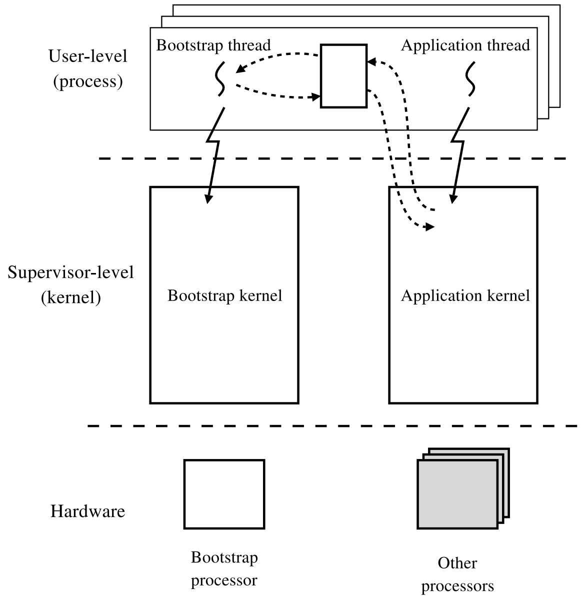

Figure 1. Overview of the application kernel approach.

Every process is modified to run two threads, a *bootstrap thread* and an *application thread*. The application thread runs the unmodified application on the application processors and the bootstrap thread runs on the bootstrap processor waiting for messages from the application kernel. For every application thread there is one bootstrap thread and each such thread pair communicates through a communication area in the process' address space. Figure 1 shows the overall approach with the bootstrap kernel and bootstrap thread shown on the left and the application kernel and application thread on the right. The communication area is shown in the center of the process.

The application runs as before except when performing operations involving the kernel, i.e. system calls and processor exceptions (for instance page faults). On such an event, the application thread traps into the application kernel which handles the event by placing a message in the communication area. The actual event will be handled at a later point by the bootstrap thread, which simply issues the same system call or exception.

With this approach, very few modifications need to be done to the original kernel or applications. Applications need to be relinked in order to start the thread pair and the bootstrap kernel must start the application kernel, which can commonly be achieved through loadable modules without kernel modification. There are some special situations that

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

require kernel modification, described in Section 4.3.

Compared to master-slave kernels, our approach tries to minimize the knowledge needed to implement a SMP port, in the best case not even requiring access to the uniprocessor kernel source code. The focus is therefore a bit different. For master-slave kernels the added SMP-complexity goes into the original kernel (albeit most of it is easily separable), whereas in our case almost all of the added complexity is kept in a separate kernel. In a sense, our approach can be seen as a more general revitalization of the master-slave idea. The approach can also be compared to single system image distributed systems such as MOSIX [2], which redirects system calls to the "unique home node" of the process, although the goals of the two approaches are quite different.

3.2. Requirements

A number of requirements are placed on the architectural support by the application kernel approach, although most common processor architectures support these. The requirements are:

1. The processor architecture must support binding of interrupts to a specific processor.
2. It must be possible to generate timer interrupts locally to processors.
3. There must be a way of retrieving the current page table address.
4. It must be possible to use different interrupt handlers on different processors.

The first requirement must be fulfilled since only the bootstrap kernel handles external interrupts in our approach. The second requirement is needed for thread scheduling on the application kernel whereas the page table address is required when creating new processes. Further, since the application kernel need to handle system calls and exceptions differently than the bootstrap kernel, the application kernel needs private system call and exception handlers. All these are fulfilled on the IA-32 architecture.

Our approach also places a number of requirements on the bootstrap kernel, as listed below. These are normally available in most operating system kernels.

1. It must be possible to extend the kernel with code running in supervisor mode.
2. The bootstrap kernel must not change or remove any page mappings from the application kernel.
3. The page table must be the same for the kernel and the current process.

The first of the kernel requirements is normally fulfilled by some sort of loadable entity, e.g. kernel modules in Linux, which allow code to be inserted into a running kernel. The second requirement is necessary since the application kernel would run in invalid memory if a page mapping is removed. Also, the application processor TLB contents would be incorrect unless the bootstrap processor notifies the other processors when remapping a page. This is also a problem for applications, as we discuss in Section 4.3. Finally, the application kernel needs to be able to retrieve the current process page table which explains the last requirement. The IA-32 architecture poses an additional limitation. At SMP startup, physical memory addresses below 1MB is needed [12], a limitation posed by the legacy 8086-mode of IA-32 processors. On IA-32 it must therefore be possible to retrieve such memory from the kernel, or use known safe memory areas (this is only needed during startup).

4. Application kernel design and implementation

We have implemented a prototype application kernel for a small in-house uniprocessor kernel. The uniprocessor kernel has been implemented to provide a test platform for porting our industrial partner's kernel and is a basic kernel with threads, processes, and IPC (inter-process communication). This kernel supports dynamic creation and termination of threads and processes and is also extensible by device drivers (currently the drivers must be added at kernel compile-time). The application kernel was implemented as a device driver for our uniprocessor kernel. In this section we describe a prototype implementation of the application kernel approach. We also describe some of the design decisions for the application kernel more in detail.

4.1. System calls and exception handling

Figure 2 shows a detailed view of system call handling in our approach. Exceptions such as page faults, file I/O, device driver interaction etc. are handled the same way. For system calls, seven steps can be identified:

1. The application thread issues a system call and traps into the application kernel. The thread is then blocked by the application kernel and removed from the ready queue.
2. The application kernel enters information about the system call, i.e. the passed parameters etc., into the communication area.
3. The bootstrap thread is scheduled at some point later and finds a new message in the communication area.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

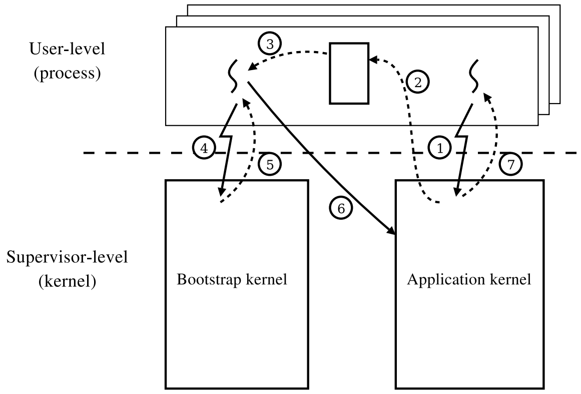

Figure 2. Execution flow when performing a system call.

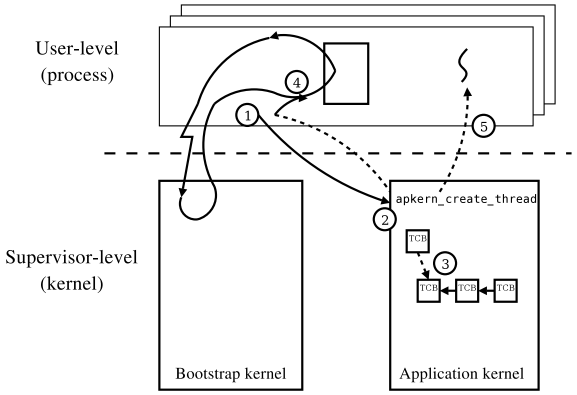

Figure 3. apkern_create_thread execution flow.

4. The bootstrap thread issues the same system call with the parameters passed in the message and traps into the bootstrap kernel.

5. The bootstrap kernel handles the system call and returns control to the bootstrap thread, which in turn stores the result of the system call in the communication area.

6. The bootstrap thread issues the **apkern_activate_thread** (described in Section 4.2) to signal that the system call has been handled.

7. Finally, the application kernel returns control to the application thread which can now continue since the system call has been handled.

System calls such as the UNIX **exit** call and exceptions that cause the process to be terminated (for example illegal instructions) are handled slightly differently 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 is therefore modified to clean up after 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 using UNIX signals), e.g. through an arriving network packet. In this case, the handler (which is installed at startup, if needed) 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.

On every system call or exception, the application kernel schedules another thread. Since the system call latency depends on the scheduling of the bootstrap thread, the application kernel could otherwise stall for a long time waiting for the call to be performed. The processor also reloads the page table on each thread switch to clear the TLB if a page fault caused the page table to be updated.

4.2. Application kernel interface

The application startup library (`crt0`) is modified for our approach. We modified `crt0` so that it (on the bootstrap processor) opens the application kernel device driver and calls the driver to add a new application thread. Thereafter a message-waiting loop of the bootstrap thread is entered. There are three main routines in the application kernel that are called from the bootstrap thread or the bootstrap kernel. These cannot (other than indirectly) be called by application threads since the functions are only available through the driver. The exported interface is shown below.

- **apkern_init**: This routine is called once on system startup, for instance on the device driver initialization call. It performs the following tasks:
    - It initializes data structures in the application kernel, including the ready-queue structure.
    - 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_create_thread**: This function is called from the thread running on the bootstrap kernel when the

```
loop
    if (communication area has entry) then
        handle_message(entry)
        apkern_activate_thread()
    else
        reschedule()

function handle_message(msg)
    if (msg is page fault) then
        force pagefault by referencing the address
    else if (msg is system call)
        cause the corresponding system call
        place result in communication area
    else if (msg is exception)
        cause the corresponding exception
        place result in communication area
```

Figure 4. Bootstrap thread loop

process is started. This function creates a new thread on the application kernel and makes it schedulable.

- **apkern_activate_thread**: This routine is called when the bootstrap thread has handled a message (and should therefore wake up an application thread). The call changes the state of the corresponding application thread from blocked to ready and inserts the TCB into the ready queue.

Figure 3 shows a detailed chain of events when **apkern_create_thread** is called. The handling is split in five stages:

1. The newly created bootstrap thread starts executing.
2. The bootstrap thread makes a call to **apkern_create_thread**.
3. The application kernel creates and initializes the TCB (thread control block) associated with the new application thread and inserts it into the ready queue.
4. The thread on the bootstrap kernel then enters a loop polling for messages in the communication area. The loop and message handling is shown in Figure 4.
5. The application thread is scheduled for execution on one of the application processors.

The logical and actual implementation structure is shown in Figure 5. As illustrated by the Figure, the bootstrap thread logically issues the **apkern_create_thread** call directly to the application kernel. Implementation-wise, communication between the bootstrap thread and the application kernel is done through a device-driver call to the bootstrap kernel. There are are therefore three steps for an **apkern_create_thread** call:

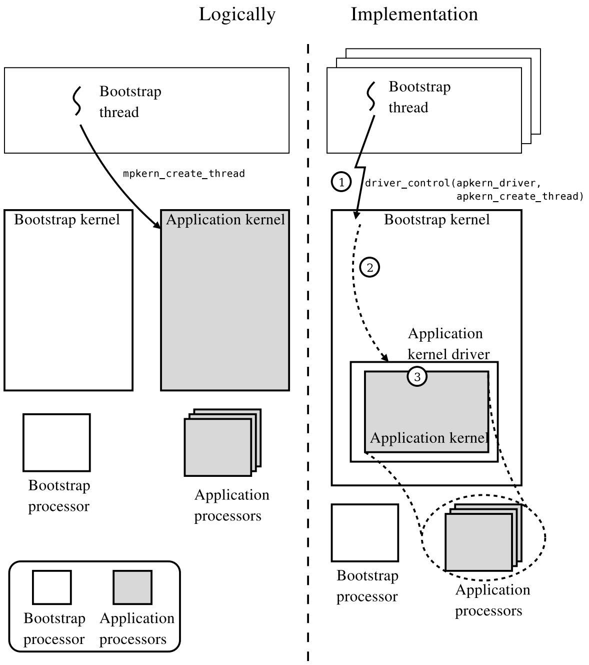

Figure 5. Implementation structure of the application kernel approach.

1. The bootstrap thread issues a **driver_control** call (in our kernel, for other systems **write** or **ioctl** might be used) to the bootstrap kernel, passing the **apkern_create_thread** operation as argument.
2. The bootstrap kernel parses the request, checks permissions etc and passes the call to the device driver-specific **driver_control** implementation.
3. Finally, the device driver performs the operation on the application kernel data structures, in this case adding a thread to the ready queue.

Note that all the steps above are performed by the device driver running on the bootstrap processor. Data structures such as the ready queue in the application kernel therefore have to be protected both from application processors and the bootstrap processor. The ready queue is protected by a spinlock in our implementation.

4.3. Limitations and extensions

A special correctness problem is asynchronous page table invalidations, i.e. page table changes not caused by the process itself, which could occur for instance if the bootstrap kernel swaps out a page to disk. The problem is that, although the bootstrap kernel updates the page table, the

TLB contents on the application processor will be inconsistent in the address space with the unmapped page. If the corresponding application thread is scheduled, it would then run in invalid memory. To handle this situation, the bootstrap kernel must be modified to send invalidation messages to the application processors, a change which should be straightforward to implement in most systems.

There is also a number of optimizations possible for the 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 [2] could then be used to move kernel-bound threads to the bootstrap processor during runtime. Further, it is also possible to implement some system calls 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 on the application kernel. We have chosen not to pursue these optimizations for the prototype, as well as implementing load balancing, process migration etc., since we deemed a proof-of-concept more important.

Another fairly simple extension is to allow multithreaded processes. In order to run multithreaded programs, we simply start a bootstrap thread for each application thread. One difference for multithreaded applications is that all threads in a process must be blocked when one of the threads generates a page fault (again in order to not execute a thread in an invalid address space), which adds some cross-processor synchronization issues.

4.4. Implementation complexity

The implementation complexity of our approach is a very important component. Our implementation consists of about 2500 lines of C and assembly code, most of which deals with SMP initialization (support libraries, like the C-library, have been excluded from these numbers). The changes at the application level is limited to a few lines of modified code to `crt0` and the implementation of the bootstrap thread loop (about 120 lines). The device driver implementation encompasses slightly more than 250 lines. We did not implement any exception handling apart from page faults in our prototype kernel, although adding exception handling should not require very much code. The handling of additional system calls also does not add any extra code to the implementation.

No changes were made to the original (bootstrap) kernel, although an implementation of TLB invalidation would be needed for the case where the bootstrap kernel asynchronously unmaps memory pages. An implementation of TLB invalidation should require fairly small amounts of work.

We believe that our approach is fairly generic, i.e. most of the application kernel code could be reused for other operating systems. The parts that are operating system-specific is the driver implementation, the startup library, system call parameter passing, some exception handling and (on IA-32) the initialization of the IDT (Interrupt Descriptor Table, the interrupt vector). We are therefore confident that the application kernel approach could be added to other operating systems without a major effort.

5. Performance evaluation

In this section, we present an evaluation of the performance of the application kernel approach in order to show under what conditions it is feasible performance-wise. The performance measurements were done under the Simics simulator [17], which simulates a complete IA-32 system with 1 to 8 processors. The implementation runs on real hardware, but the Simics simulator is a cost-effective and flexible tool for running automated tests on many different hardware configurations.

It should be noted that Simics does not implement instruction timing accurately on IA-32, so the base for the measurements is the number of instructions executed and not clock cycles. Further, other performance issues relevant in multiprocessor systems [8], such as costs associated with data alignment, cross-processor cache access etc. is not accounted for. However, we believe that these measurements still gives an accurate enough indication of real-world performance for the approach.

We have performed two kinds of microbenchmarks on our approach. First we evaluated the added latency introduced to system calls and exceptions. Second, we measured the throughput as the effective work performed by processors under varying system load.

5.1. Latency

We first performed a benchmark of system call latency with our approach. Exception handling (for instance page faults) would produce similar latencies since it works by the same principles. We measured latency by adding inspection points before and after a system call (**gettid**, which only returns the thread id of the current thread) in an empty system. The results are an average of 1000 measurement iterations.

Since the application thread remains blocked while the bootstrap thread handles the system call, the latency of system calls depends on three factors. First, the bootstrap thread must be scheduled and issue the system call. Thereafter, the bootstrap kernel must handle the system call. Finally, the application kernel must reschedule the application thread (the thread was entered into the ready queue through a call to the device driver made by the bootstrap

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

thread). The worst-case system call latency therefore occurs when the processes in the system are completely CPU-bound, since the application thread scheduling then depends completely on the ready queue length and timer interval.

For our uniprocessor kernel, the **gettid** system call takes about 100 instructions to complete (this system call never switches thread). With the application kernel, the latency added using our approach is about 1200 instructions. The latency is not dependent on the number of processors since the main delay occurs on the bootstrap processor (the application processors spin until some thread becomes schedulable). From the latency measurements, we conclude that our approach is not feasible for applications and systems with many system calls or requirements on short response time.

5.2. Throughput

We evaluated two factors in our throughput measurements. First we measured the number of loops a thread iterated during a given time (totally there are 20 single-threaded loop-processes in the system). In the loop, the application executes a fixed amount of user-level instructions after which it performs a system call. The system call in turn spins in the kernel for a configurable number of instructions. This measurement gives an indication of system throughput in terms of how much work the collection of threads perform during a given time.

We also measured the proportion of non-idle time in the bootstrap thread, i.e. the proportion of bootstrap thread loops that contained message handling. This measurement shows the room for further scalability improvements, or in other words gives the point when adding more processors not will give any performance increase.

Both the number of user and kernel instructions in our tests were configurable. We ran the tests at three levels of kernel load, 151, 451, and 4510 instructions per system call. The number of user instructions were then selected to provide user/kernel proportions of 80, 90, 92.5, 95, 97 and 99%. These tests show the performance at frequent, relatively short system invocations, fairly long system calls and infrequent time-consuming system calls at different levels of user/kernel execution.

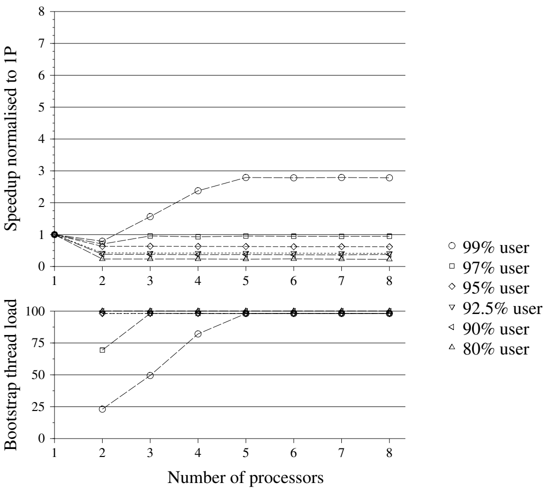

Figure 6. Speedup and bootstrap kernel load at 151-instruction system calls.

Figures 6, 7 and 8 show the results of the throughput measurements for the three levels of kernel load, 151, 451 and 4510 instructions per system call respectively. The upper part of the figures illustrate the speedup gained by adding additional processors, normalized to the uniprocessor performance. The lower part shows the load on the bootstrap processor. From the figures we can see that the approach scales fairly well for long system calls. The figures also show that the potential for scalability increases with longer, less frequent system calls. Also, Figure 8 illustrates that even two processors (i.e. one application processor) can give performance increases. This is because the bootstrap kernel handles kernel calls while the application kernel runs user code in this case. Further, the figures show that the maximum throughput using our approach is limited by the capacity of the bootstrap processor. It turns out that the maximum capacity of the bootstrap processor can be written as

$$C = ips = scps * (ipsc + opsc)$$

Where C is the capacity, ips the number of instructions per second, $scps$ the system calls per second, $ipsc$ the average number of instructions per system call and $opsc$ is the overhead in instructions per system call.

We know the capacity in terms of instructions per second and by looking at the values in figures 6, 7 and 8 we can calculate the average overhead per system call added by our approach. From the curves for 99% in Figure 6, 97% in Figure 7, and 90% in Figure 8 we get an overhead per system call of about 2000 instructions.

The equation above makes it possible to calculate the maximum throughput for arbitrary application program behavior. For instance, if we have a case where a system call takes 1000 instructions on average and the application program issues one system call every 15,000th instruction on average, we know that the maximum number of system calls per second is (if we assume 100,000,000 instructions per second)

$$\frac{ips}{(ipsc + opsc)} = \frac{100 * 10^6}{(1000 + 2000)} \approx 30000$$

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

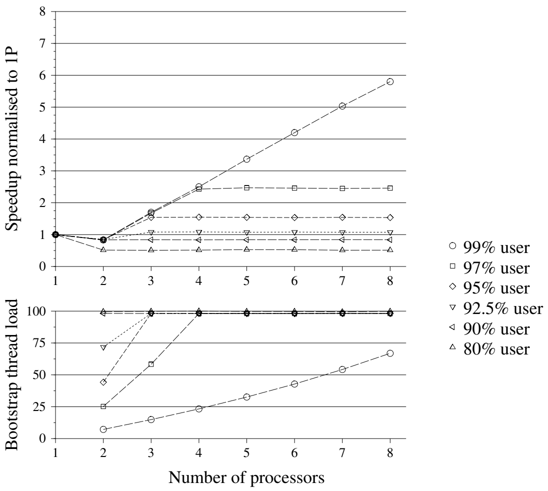

Figure 7. Speedup and bootstrap kernel load at 451-instruction system calls.

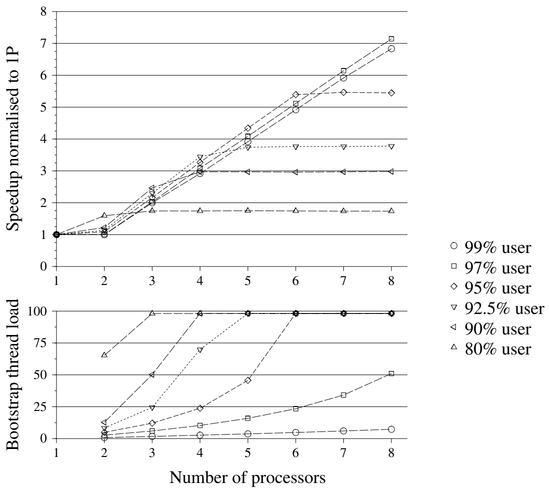

Figure 8. Speedup and bootstrap kernel load at 4510-instruction system calls.

The number of system calls per second for the uniprocessor case is $100 * 10^6/15000$ and the maximum speedup is thus

$$\frac{100 * 10^6/(1000 + 2000)}{100 * 10^6/15000} = \frac{15000}{1000 + 2000} \approx 5$$

In a highly optimized production version of our approach it is reasonable to expect that we would be able to decrease the overhead per system call even more, and thus obtain better performance. The above equation makes it possible to quantify the performance increase as a function of the implementation optimizations leading to lower overhead.

6. Conclusion

In this paper, we have presented an novel approach for adding SMP support to a uniprocessor operating system. Our approach provides an alternative with less implementation complexity than the traditional approaches, carrying similarities to both traditional master-slave systems and distributed systems. In some cases, our approach does not require any modification at all of the uniprocessor kernel, while some minor changes are required in other cases.

There are several advantages with our approach. First, we do not need to go into 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 SMP version. We also expect that a large portion of the effort of writing the application kernel can be reused for other uniprocessor kernels.

The disadvantage with the application approach is limited performance. However, our microbenchmarks show that our approach can provide good speedup for applications with a low frequency of system calls. For applications with frequent and short system calls, the performance gains do not show up until we use more than two processors. However, systems with time-consuming kernel invocation show performance gains also for two processors. We also provide a formula that makes it possible to quantify the user perceived performance in terms of throughput as a function of the overhead per system call added by our approach.

Acknowledgments

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

References

[1] A. C. Arpaci-Dusseau and R. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 43–56, 2001.

[2] 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.

[3] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese.

Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, Vancouver, Canada, June 2000.

[4] 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.

[5] 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.

[6] J. M. Denham, P. Long, and J. A. Woodward. DEC OSF/1 symmetric multiprocessing. *Digital Technical Journal*, 6(3), 1994.

[7] F. des Places, N. Stephen, and F. Reynolds. Linux on the OSF Mach3 microkernel. In *Proceedings of the Conference on Freely Distributable Software*, February 1996.

[8] B. Gamsa, O. Krieger, E. W. Parsons, and M. Stumm. Performance issues for multiprocessor operating systems. Technical report, Computer Systems Research Institute, university of Toronto, November 1995.

[9] 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.

[10] 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.

[11] 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.

[12] Intel Corporation. *MultiProcessor Specification, version 1.4*. Intel Corporation, May 1997.

[13] J. Kahle. Power4: A dual-CPU processor chip. In *Proceedings of the 1999 International Microprocessor*, San Jose, CA, October 1999.

[14] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, and G. Skinner. Symmetric multiprocessing in Solaris 2.0. In *Compcon*, pages 181–186. IEEE, 1992.

[15] G. Lehey. Improving the FreeBSD SMP implementation. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 155–164. USENIX, June 2001.

[16] 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.

[17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[18] D. 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.

[19] S. Muir and J. Smith. AsyMOS - an asymmetric multiprocessor operating system. In *Proceedings of OPENARCH '98*, pages 25–34, April 1998.

[20] QNX Software Systems Ltd. The QNX neutrino microkernel, 2003. http://qdn.qnx.com.

[21] T. Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, Queens' College, University of Cambridge, April 1995.

[22] C. Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, Boston, first edition, 1994.

[23] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, November 2000.

[24] K. Yaghmour. A practical approach to linux clusters on SMP hardware, http://www.opersys.com, July 2002.

[25] 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.