

Performance and Implementation Complexity in Multiprocessor Operating System Kernels

Simon Kågström

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

<http://www.ipd.bth.se/ska>



IN PARTNERSHIP WITH THE

Knowledge Foundation



- This work has been performed as a subproject in the BESQ project
- Funded by Ericsson together with the KK foundation
- Advisors: Prof. Lars Lundberg and Dr. Håkan Grahn



BESQ - Blekinge Engineering Software Qualities

IN PARTNERSHIP WITH THE

Knowledge Foundation



1 Background

- Operating System Development
- Research Questions

2 Papers

- Paper I – Multiprocessor OS Survey
- Paper II – Traditional MP Port
- Paper III – Application Kernel
- Paper IV – Program Instrumentation

3 Conclusions and future work

1 Background

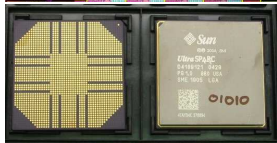
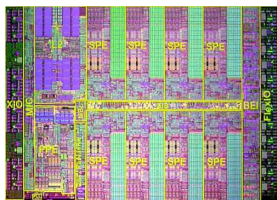
- Operating System Development
- Research Questions

2 Papers

- Paper I – Multiprocessor OS Survey
- Paper II – Traditional MP Port
- Paper III – Application Kernel
- Paper IV – Program Instrumentation

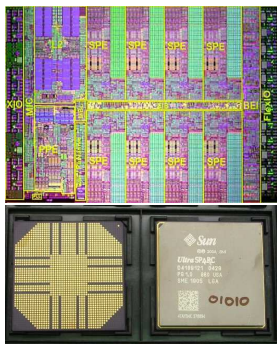
3 Conclusions and future work

Processor Trends



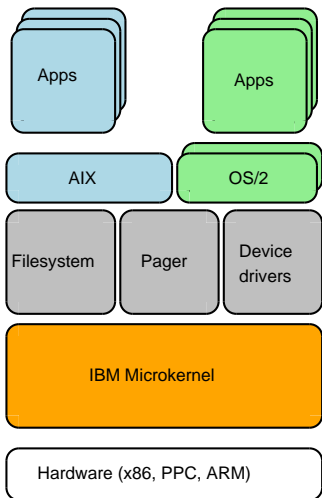
- The CPU industry is currently moving towards more multiprocessing
 - Chip multiprocessing and simultaneous multithreading
- The pictures show the IBM/Sony Cell and the Sun Niagara chips

Processor Trends



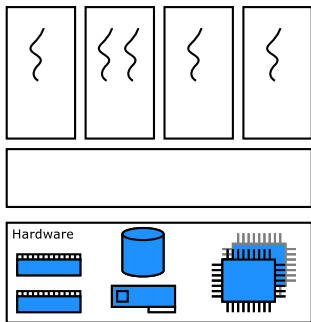
- The CPU industry is currently moving towards more multiprocessing
 - Chip multiprocessing and simultaneous multithreading
- The pictures show the IBM/Sony Cell and the Sun Niagara chips
- It will be increasingly important for software to handle multiprocessors in the future

Workplace OS



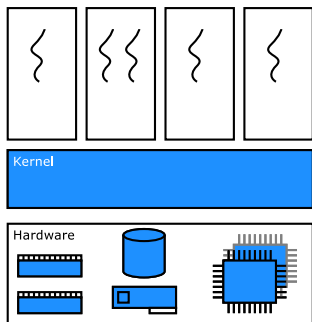
- Workplace OS was a very large IBM operating system project started in 1991
- Workplace OS was going to unify **most** IBM operating systems
 - It would run on everything from PDAs to large server systems
 - It would support multiple personalities concurrently
- Yet quite unknown to most people, why?
- After five years of development, the project was canceled

Operating System Background



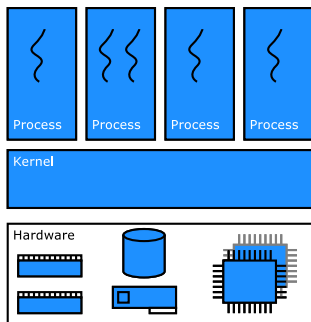
- **Hardware**
 - Processors, network interfaces etc.
- **Kernel**
 - Hardware access, multiplexing
 - Needs to be protected against concurrent access from multiple processors
- **Processes**
 - Resource containers (files etc.)
 - Often also address space
- **Threads**
 - Execution context
- **Virtual address spaces**
 - A virtualized view of the computer memory
 - Mapping virtual $\langle - \rangle$ physical pages

Operating System Background



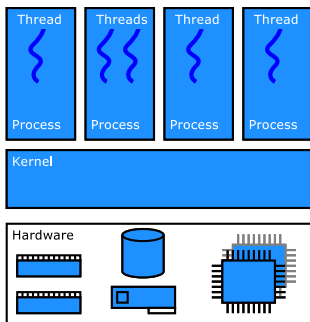
- Hardware
 - Processors, network interfaces etc.
- **Kernel**
 - Hardware access, multiplexing
 - Needs to be protected against concurrent access from multiple processors
- Processes
 - Resource containers (files etc.)
 - Often also address space
- Threads
 - Execution context
- Virtual address spaces
 - A virtualized view of the computer memory
 - Mapping virtual $\langle - \rangle$ physical pages

Operating System Background



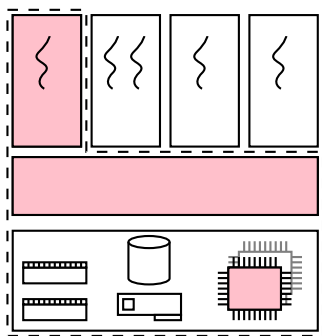
- Hardware
 - Processors, network interfaces etc.
- Kernel
 - Hardware access, multiplexing
 - Needs to be protected against concurrent access from multiple processors
- Processes
 - Resource containers (files etc.)
 - Often also address space
- Threads
 - Execution context
- Virtual address spaces
 - A virtualized view of the computer memory
 - Mapping virtual $\langle - \rangle$ physical pages

Operating System Background



- Hardware
 - Processors, network interfaces etc.
- Kernel
 - Hardware access, multiplexing
 - Needs to be protected against concurrent access from multiple processors
- Processes
 - Resource containers (files etc.)
 - Often also address space
- **Threads**
 - Execution context
- Virtual address spaces
 - A virtualized view of the computer memory
 - Mapping virtual $\langle - \rangle$ physical pages

Operating System Background



Address space



- Hardware
 - Processors, network interfaces etc.
- Kernel
 - Hardware access, multiplexing
 - Needs to be protected against concurrent access from multiple processors
- Processes
 - Resource containers (files etc.)
 - Often also address space
- Threads
 - Execution context
- **Virtual address spaces**
 - A virtualized view of the computer memory
 - Mapping virtual $\langle - \rangle$ physical pages

- There is one primary research question (PQ) and three secondary research questions (RQ1, RQ2, RQ3)
- Primary research question (PQ):

How can we find a good tradeoff between performance and development effort when porting a uniprocessor operating system kernel to a multiprocessor and how can we evaluate it in an efficient way?

- Research Question 1 (RQ1):

What is the cost and performance benefit of performing a traditional symmetric lock-based multiprocessor port of an operating system kernel?

- Research Question 2 (RQ2):

What are the lower limits of development effort when performing a multiprocessor port of an operating system kernel? Can multiprocessor support be added to the operating system without modifying the original kernel?

- Research Question 3 (RQ3):

How is it possible to lower the perturbation caused by program instrumentation?

- Paper I: *Scalability vs. Development Effort for Multiprocessor Operating System Kernels*, submitted to the Journal of Systems and Software
- Paper II: *The Design and Implementation of Multiprocessor Support for an Industrial Operating System Kernel*, submitted to the Journal of Systems Architecture
- Paper III: *The Application Kernel Approach - a Novel Approach for Adding SMP Support to Uniprocessor Operating Systems*, submitted to Software - Practice and Experience
- Paper IV: *Automatic Low Overhead Program Instrumentation with the LOPI framework*, published at the Workshop on Interaction between Compilers and Computer Architectures, 2005

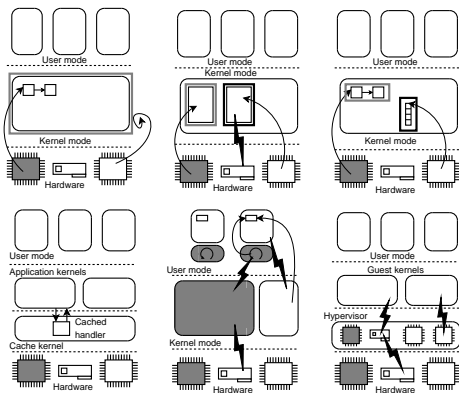
- 1 Background
 - Operating System Development
 - Research Questions
- 2 Papers
 - Paper I – Multiprocessor OS Survey
 - Paper II – Traditional MP Port
 - Paper III – Application Kernel
 - Paper IV – Program Instrumentation
- 3 Conclusions and future work

- 1 Background
 - Operating System Development
 - Research Questions
- 2 Papers
 - Paper I – Multiprocessor OS Survey
 - Paper II – Traditional MP Port
 - Paper III – Application Kernel
 - Paper IV – Program Instrumentation
- 3 Conclusions and future work

Overview of Paper I

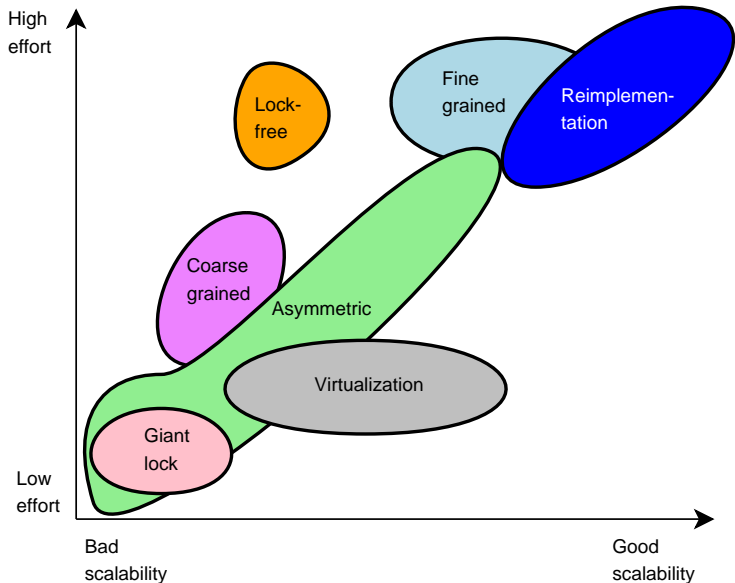
- Paper I presents a survey of multiprocessor operating system implementations
- The paper also contains a case study of the multiprocessor evolution of the Linux kernel
- Contributions:
 - ① Categorization of existing operating systems into 7 implementation techniques
 - ② Comparison of development effort and performance for the techniques
 - ③ Case-study of Linux multiprocessor development

Categorization of Multiprocessor OSeS



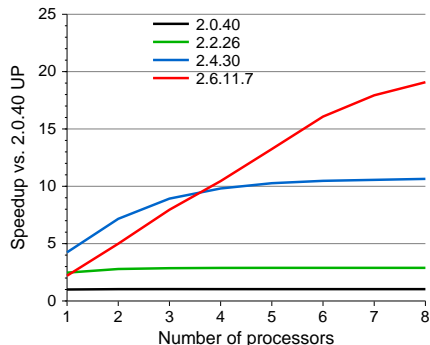
- 22 OSeS were categorized:
- Giant locking (3)
- Coarse-grained locking (2)
- Fine-grained locking (6)
- Lock-free approaches (2)
- Asymmetric approaches (3)
- Virtualization (5)
- Reimplementation (1)

Design Space



- The Linux multiprocessor implementation has gone through a transition from giant locking to fine-grained locking
- 2.0: giant locking
- 2.2: coarse grained locking
- 2.4: fine grained locking
- 2.6: fine grained locking (refined)
- Performance and implementation was compared between the different versions
- We evaluated performance with the `postmark` benchmark

Linux Case Study, Results



- The uniprocessor performance has improved greatly since 2.0
- 2.4 has the best uniprocessor performance
- 2.0 and 2.2 does not scale at all for this benchmark
- 2.6 has very good scalability

Linux Case Study, Results II

Version	Number of lock operations					sema	code size
	BKL	spinlock	rwlock	seqlock	rcu		
2.0.40	17	0	0	0	0	49	45,770
2.2.26	226	329	121	0	0	121	53,281
2.4.30	193	989	300	0	0	332	65,552
2.6.11.7	101	1,717	349	56	14	650	105,846

- The number of lock operations in the kernel reflects the implementation strategy
- 2.0 has only one lock, taken in very few places
- Newer kernels relaxed the giant lock

Linux Case Study, Results II

Version	Number of lock operations					sema	code size
	BKL	spinlock	rwlock	seqlock	rcu		
2.0.40	17	0	0	0	0	49	45,770
2.2.26	226	329	121	0	0	121	53,281
2.4.30	193	989	300	0	0	332	65,552
2.6.11.7	101	1,717	349	56	14	650	105,846

- The number of lock operations in the kernel reflects the implementation strategy
- 2.0 has only one lock, taken in very few places
- Newer kernels relaxed the giant lock

Linux Case Study, Results II

Version	Number of lock operations					sema	code size
	BKL	spinlock	rwlock	seqlock	rcu		
2.0.40	17	0	0	0	0	49	45,770
2.2.26	226	329	121	0	0	121	53,281
2.4.30	193	989	300	0	0	332	65,552
2.6.11.7	101	1,717	349	56	14	650	105,846

- The number of lock operations in the kernel reflects the implementation strategy
- 2.0 has only one lock, taken in very few places
- Newer kernels relaxed the giant lock

Linux Case Study, Results II

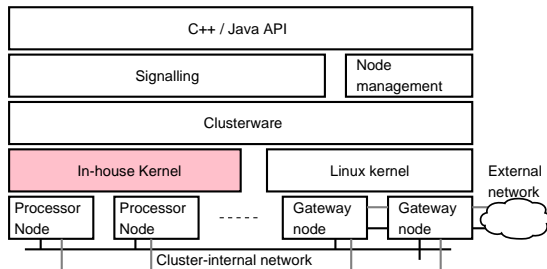
Version	Number of lock operations					sema	code size
	BKL	spinlock	rwlock	seqlock	rcu		
2.0.40	17	0	0	0	0	49	45,770
2.2.26	226	329	121	0	0	121	53,281
2.4.30	193	989	300	0	0	332	65,552
2.6.11.7	101	1,717	349	56	14	650	105,846

- The number of lock operations in the kernel reflects the implementation strategy
- 2.0 has only one lock, taken in very few places
- Newer kernels relaxed the giant lock

- 1 Background
 - Operating System Development
 - Research Questions
- 2 Papers
 - Paper I – Multiprocessor OS Survey
 - **Paper II – Traditional MP Port**
 - Paper III – Application Kernel
 - Paper IV – Program Instrumentation
- 3 Conclusions and future work

- Paper II presents a traditional multiprocessor operating system port
- The port was performed on an industrial operating system kernel
- We used a traditional lock-based approach for protection of the operating system kernel
- Contributions from this paper:
 - ① Technical solutions to common problems in multiprocessor operating system ports
 - ② Discussion of implementation experiences from the port

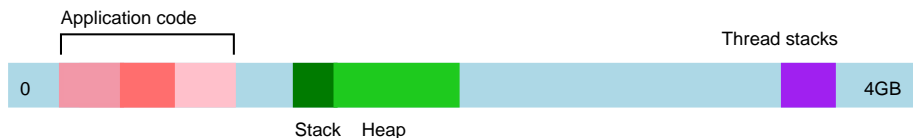
The Industrial Operating System Kernel



- Distributed, fault-tolerant cluster system
- Distributed RAM-resident database
- Mainly used in telecom systems

The Industrial Operating System Kernel, II

- The operating system is based around three basic entities:
 - **Containers**: address spaces
 - **Processes**: resource containers
 - **Threads**: execution context
- The programming model relies on fast processes:
 - Efficient address space handling, small memory requirements
 - Applications are always mapped in memory
 - No paging to disk

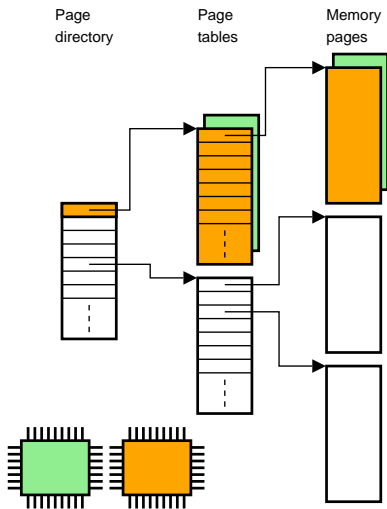


Address space

Overview of the Multiprocessor Port

- A giant locking scheme was used for the port
 - A single lock protects the entire kernel
 - Similar to Linux 2.0
- Hardware interrupts (except timer interrupts) are routed to one processor
- Portions of the code, e.g., floating point support and processor startup needed to be modified to support multiprocessors
- The reason for the simple approach was to simplify the first port and provide a ground to improve the port later

Processor-local data



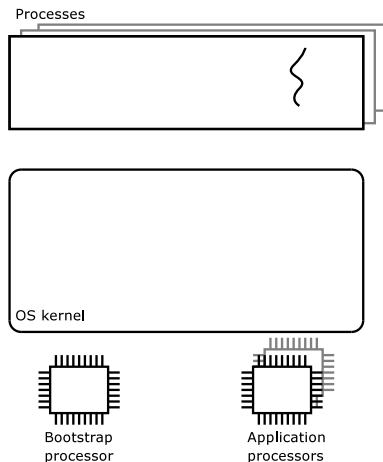
- An area of virtual memory is reserved for processor-local variables
- This allows processor-local variables to be accessed exactly as before
- Large structures, e.g., the kernel stack, need indirection
- Processor-local data is a problem with multithreaded address spaces - both processors would use the same locals
 - This is solved by a lazy method where the address space is split when two or more processors execute in it

- The operating system has a large and complex code base
 - Around 2.5M lines of code totally
 - Around 160,000 lines were relevant for the multiprocessor port
- Only around 1% of the relevant code base has been modified
- The project took around two years to implement
 - One developer, part-time
 - Specialized operating system, complex to setup and configure
 - Working off-site complicated interaction with the developers
- The giant locking approach limits performance
 - Around 20% improvement over the uniprocessor performance
- This suggests the need for alternative approaches

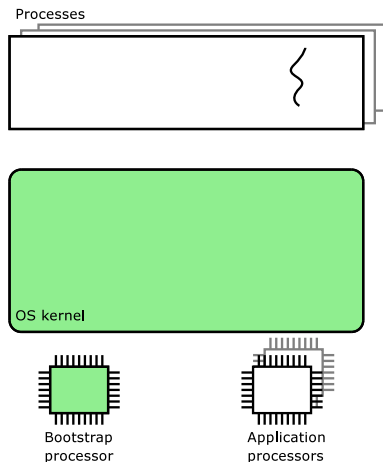
- The operating system has a large and complex code base
 - Around 2.5M lines of code totally
 - Around 160,000 lines were relevant for the multiprocessor port
- Only around 1% of the relevant code base has been modified
- The project took around two years to implement
 - One developer, part-time
 - Specialized operating system, complex to setup and configure
 - Working off-site complicated interaction with the developers
- The giant locking approach limits performance
 - Around 20% improvement over the uniprocessor performance
- This suggests the need for alternative approaches

- 1 Background
 - Operating System Development
 - Research Questions
- 2 Papers
 - Paper I – Multiprocessor OS Survey
 - Paper II – Traditional MP Port
 - **Paper III – Application Kernel**
 - Paper IV – Program Instrumentation
- 3 Conclusions and future work

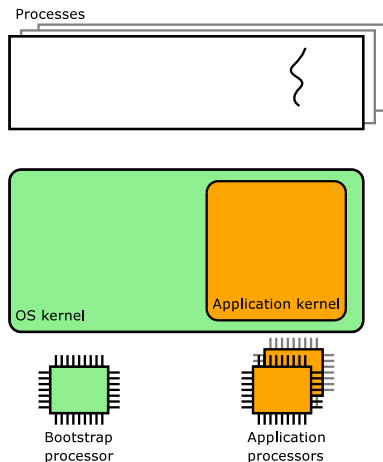
- Paper III presents an alternative multiprocessor porting approach, the *application kernel approach*
- The approach has been implemented for the Linux kernel
- Contributions:
 - ① This paper demonstrates that it is feasible to implement the application kernel approach in a real setting
 - ② The paper also shows that the implementation effort can be kept low and that the performance is comparable to standard Linux for compute-bound tasks



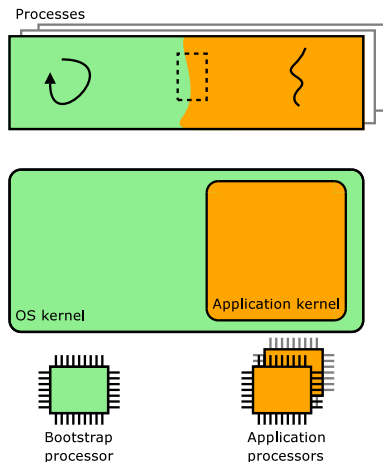
- Two kernels:
 - The original uniprocessor kernel (the bootstrap kernel, Linux in this case)
 - A small generic kernel (the Application kernel)
- The bootstrap kernel runs on one processor while the application kernel controls the rest
- Each process has two threads: an application thread and a bootstrap thread
- Both kernels have separate interrupt handlers



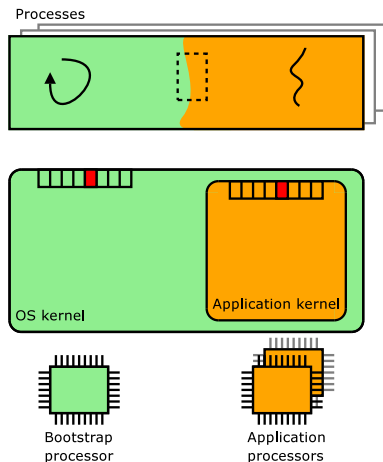
- Two kernels:
 - The original uniprocessor kernel (the **bootstrap kernel**, Linux in this case)
 - A small generic kernel (the Application kernel)
- The bootstrap kernel runs on one processor while the application kernel controls the rest
- Each process has two threads: an application thread and a bootstrap thread
- Both kernels have separate interrupt handlers



- Two kernels:
 - The original uniprocessor kernel (the bootstrap kernel, Linux in this case)
 - A small generic kernel (the **Application kernel**)
- The bootstrap kernel runs on one processor while the application kernel controls the rest
- Each process has two threads: an application thread and a bootstrap thread
- Both kernels have separate interrupt handlers

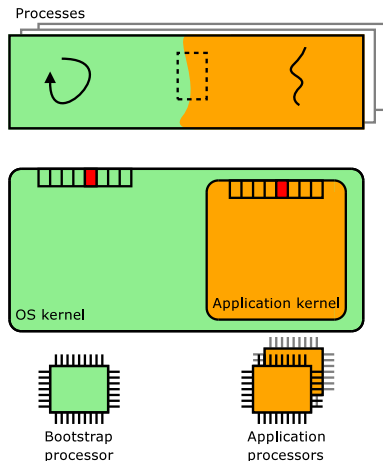


- Two kernels:
 - The original uniprocessor kernel (the bootstrap kernel, Linux in this case)
 - A small generic kernel (the Application kernel)
- The bootstrap kernel runs on one processor while the application kernel controls the rest
- Each process has two threads: an **application thread** and a **bootstrap thread**
- Both kernels have separate interrupt handlers



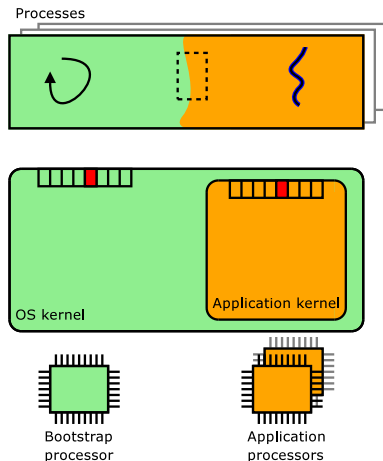
- Two kernels:
 - The original uniprocessor kernel (the bootstrap kernel, Linux in this case)
 - A small generic kernel (the Application kernel)
- The bootstrap kernel runs on one processor while the application kernel controls the rest
- Each process has two threads: an application thread and a bootstrap thread
- Both kernels have **separate interrupt handlers**

Application Kernel Example



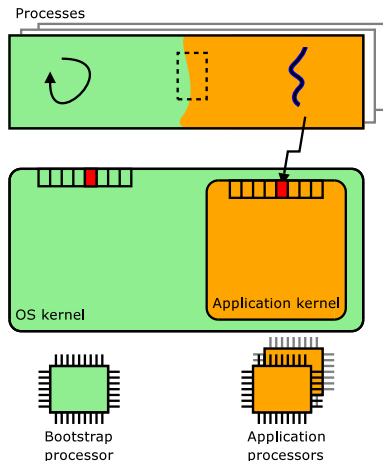
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 The bootstrap thread performs the corresponding system call
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Example



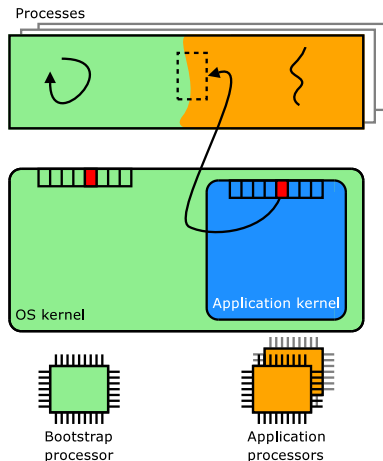
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 The bootstrap thread performs the corresponding system call
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Example



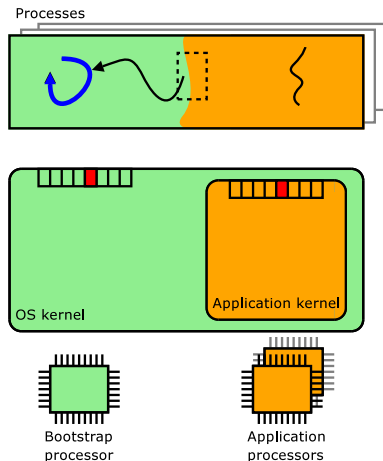
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 The bootstrap thread performs the corresponding system call
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Example



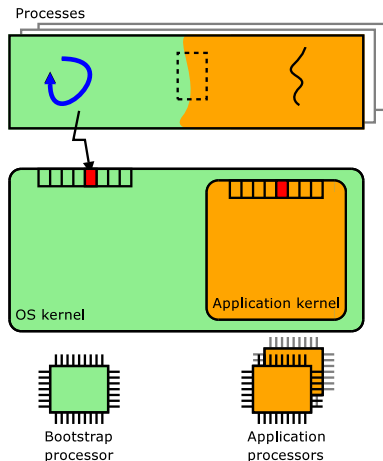
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 The bootstrap thread performs the corresponding system call
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Example



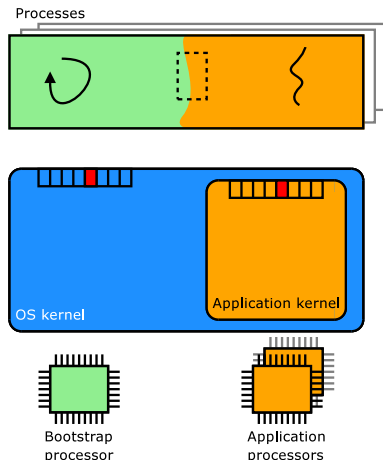
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 **The bootstrap thread finds the entry in the shared area**
- 5 The bootstrap thread performs the corresponding system call
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Example



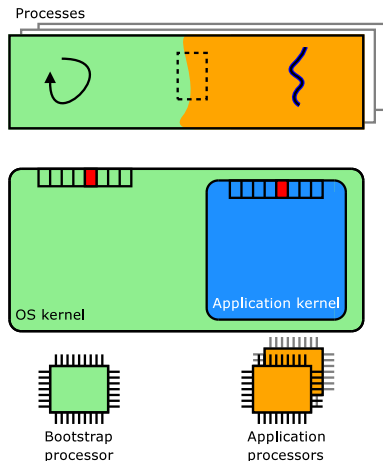
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 **The bootstrap thread performs the corresponding system call**
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Example



- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 The bootstrap thread performs the corresponding system call
- 6 **The bootstrap kernel handles the system call**
- 7 The application kernel schedules the thread again

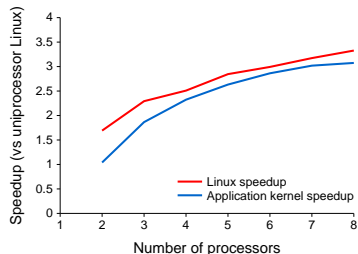
Application Kernel Example



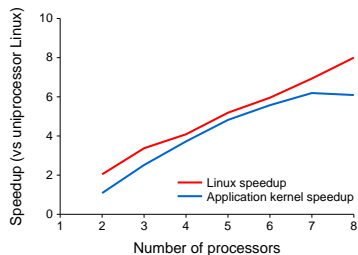
- 1 An application performs a system call
- 2 The call is handled by the application kernel
- 3 The application kernel the call into the shared area
- 4 The bootstrap thread finds the entry in the shared area
- 5 The bootstrap thread performs the corresponding system call
- 6 The bootstrap kernel handles the system call
- 7 The application kernel schedules the thread again

Application Kernel Evaluation, performance

SPLASH2 FFT



SPLASH2 RADIX

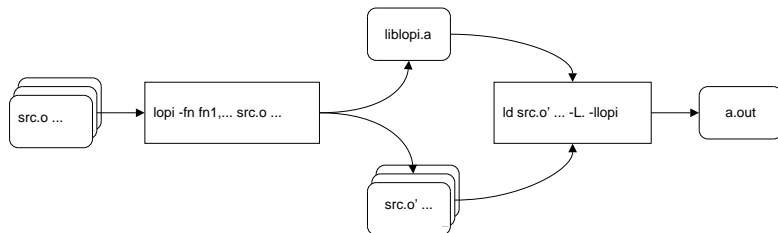


- Kernel invocations are more expensive with the application kernel approach
 - An “empty” system call takes roughly five times more time
- Computation-bound programs still benefit greatly
 - The SPLASH 2 benchmarks has similar performance to native Linux

- 1 Background
 - Operating System Development
 - Research Questions
- 2 Papers
 - Paper I – Multiprocessor OS Survey
 - Paper II – Traditional MP Port
 - Paper III – Application Kernel
 - **Paper IV – Program Instrumentation**
- 3 Conclusions and future work

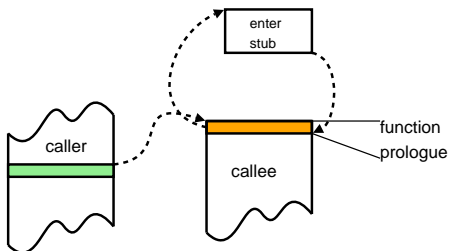
- This paper presents the LOPI framework for binary applications instrumentation
- Instrumentation can be used for performance measurement, security, debugging, code coverage analysis etc.
- LOPI tries to minimize the execution time and perturbation of instrumented code
- Main contribution:
 - 1 Providing optimized implementations for common instrumentation cases

The LOPI approach



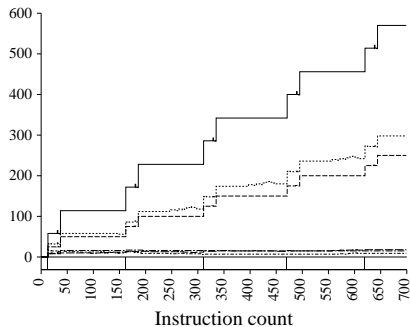
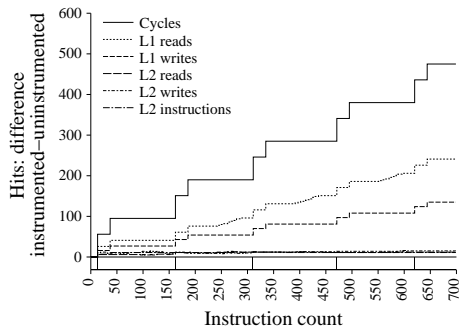
- LOPI is a package which provides a low-level interface to instrumenting applications
- The package is general and the user provides the type of instrumentation performed
- The package currently provides optimized instrumentation of function entry and exit

LOPI Optimizations



- Instead of generating new stubs for each instrumentation point, LOPI reuses stubs whenever possible
- LOPI also uses a special case for the common case of stack setup
- LOPI performs the instrumentation of function returns during runtime by overwriting the return address
- The return stubs are ordered as a stack to encourage reuse

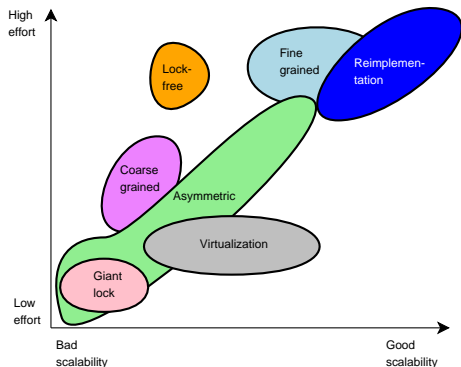
Evaluation



- We compared LOPI to the Dyninst instrumentation library
- Our evaluation shows fewer instructions and fewer cache accesses than Dyninst, at the cost of slightly more branch prediction misses

- 1 Background
 - Operating System Development
 - Research Questions
- 2 Papers
 - Paper I – Multiprocessor OS Survey
 - Paper II – Traditional MP Port
 - Paper III – Application Kernel
 - Paper IV – Program Instrumentation
- 3 Conclusions and future work

Conclusions



- PQ (tradeoff between performance and development effort)
 - A large design space of possible implementations with different tradeoffs
- RQ1 (traditional MP kernels)
 - Traditional porting methods can be beneficial but also require a significant effort
- RQ2 (multiprocessor support without modifying the original kernel)
 - The application kernel approach shows that multiprocessor support can be added with minimal effort
- RQ3 (instrumentation perturbation)
 - The LOPI tool shows a number of low-level optimizations for program instrumentation

- The application kernel:
 - Generalization of the application kernel approach
 - Optimizations of the application kernel implementation
- The industrial operating system:
 - Relaxation of the giant lock
 - Virtualization techniques to use a large multiprocessor as a cluster
- Support tools:
 - Simulator-assisted operating system kernel debugging

- The application kernel:
 - Generalization of the application kernel approach
 - Optimizations of the application kernel implementation
- The industrial operating system:
 - Relaxation of the giant lock
 - **Virtualization techniques to use a large multiprocessor as a cluster**
- Support tools:
 - Simulator-assisted operating system kernel debugging

Thank you

Thank you for listening!