Performance and Implementation Complexity in Multiprocessor Operating System Kernels

Simon Kågström

Jacket photo: © 2005 Charlie Svahnberg

© 2005 Simon Kågström
Department of Systems and Software Engineering
School of Engineering
Publisher: Blekinge Institute of Technology
Printed by Kaserntryckeriet, Karlskrona, Sweden 2005
ISBN 91-7295-074-9

Blekinge Institute of Technology
Licentiate Dissertation Series No. 2005:15
ISSN 1650-2140
ISBN 91-7295-074-9

Performance and Implementation Complexity in Multiprocessor Operating System Kernels

Simon Kågström

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

Contact information:

Simon Kågström
Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
P.O. Box 520
372 25 Ronneby
Sweden

email: simon.kagstrom@bth.se

Listen, lad. I built this kingdom up from nothing. When I started here, all there was was swamp. Other kings said I was daft to build a castle on a swamp, but I built it all the same, just to show 'em. It sank into the swamp. So, I built a second one. That sank into the swamp. So, I built a third one. That burned down, fell over, then sank into the swamp, but the fourth one... stayed up! And that's what you're gonna get, lad: the strongest castle in these islands.

From Scene 14, The Holy Grail by Monty Python

Abstract

The increasing use of multiprocessor computers require operating system adaptations to take advantage of the computing power. However, porting an operating system kernel to run on a multiprocessor can be very difficult because of a large code base, concurrency issues when dealing with multiple threads of execution, and limited tool support for development. Likewise, it can be difficult to obtain good performance from a ported operating system without sufficient parallelism in the operating system kernel.

This thesis examines the tradeoff between performance and implementation complexity in multiprocessor operating system ports and is based on four papers. The first paper is a survey of existing multiprocessor implementation approaches and focuses on the tradeoff between performance and implementation effort. The second paper describes experiences from performing a traditional lock-based multiprocessor port while the third paper presents an alternative porting approach which aims to minimize implementation complexity. The fourth paper, finally, presents a tool for efficient instrumentation of programs, which can be used during the development of large software systems such as operating system kernels.

The main contribution of this thesis is an in-depth investigation into the techniques used when porting operating systems to multiprocessors, focusing on implementation complexity and performance. The traditional approach used in the second paper required longer development time than expected, and the alternative approach in the third paper can therefore be preferable in some cases. The need for efficient tools is also illustrated in the fourth paper.

Acknowledgments

I would first like to thank my supervisors, *Professor Lars Lundberg* and *Dr. Håkan Grahn* for their help and guidance in my PhD. studies. Without their help this thesis would never have got this far.

I am also indebted to the people at Ericsson in Älvsjö, primarily *Hans Nordebäck* and *Lars Hennert*, for giving me the opportunity to work in a real-world industrial project and providing help all the times I got stuck when working in the project.

All my *colleagues*, especially in the *PAARTS* group and the *BESQ* project have been very helpful during my work, both as friends and discussion partners. The ones I talk most with in the coffee room are *Lawrence Henesey*, *Piotr Tomaszewski*, *Johanna Törnquist*, *Mirosław Staroń*, *Kamilla Klonowska*, ዳዊት መንግስቱ (*Dawit Mengistu*), ደረጀ ጌታነህ (*Dereje Getaneh*) and *Charlie Svahnberg*. Thank you all very much.

Finally, I would like to thank *Linda Ramstedt* for being my loved one and supporting me throughout the work on this thesis.

This work has been funded by Ericsson in Älvsjö together with The Knowledge Foundation in Sweden under a research grant for the project "Blekinge – Engineering Software Qualities (BESQ)" (`http://www.ipd.bth.se/besq`).

List of Papers

The following papers are included in the thesis

- Paper I: S. Kågström, H. Grahn, and L. Lundberg. Scalability vs. Development Effort for Multiprocessor Operating System Kernels. Submitted for journal publication, August 2005.
- Paper II: S. Kågström, H. Grahn, and L. Lundberg. The Design and Implementation of Multiprocessor Support for an Industrial Operating System Kernel. Submitted for journal publication, June 2005.
- Paper III: S. Kågström, L. Lundberg, and H. Grahn. The application kernel approach - a novel approach for adding SMP support to uniprocessor operating systems. To appear in Software - Practice and Experience.
- Paper IV: S. Kågström, H. Grahn, and L. Lundberg. Automatic low overhead program instrumentation with the LOPI framework. In *Proceedings of the 9th Workshop on Interaction between Compilers and Computer Architectures*, San Francisco, CA, USA, February 2005.

The following papers are not included in the thesis.

- Paper V: 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.

Paper VI: S. Kågström, H. Grahn, and L. 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.

Paper V is an earlier version of Paper III and Paper VI is an earlier version of Paper II.

Contents

Abstract **i**

Acknowledgments **iii**

List of Papers **v**

1 Introduction **1**

1.1 Background 1

1.2 Research Questions 4

1.3 Research Methodology 5

1.3.1 Research Methods 5

1.4 Contributions in this Thesis 8

1.4.1 Chapter 2 (Paper I) 8

1.4.2 Chapter 3 (Paper II) 9

1.4.3 Chapter 4 (Paper III) 10
1.4.4 Chapter 5 (Paper IV) 11
1.5 Validity of the Results 11
1.5.1 Generalizability, External Validity 12
1.5.2 Internal Validity 13
1.6 Related Work 14
1.6.1 Traditional Multiprocessor Ports 15
1.6.2 Alternative Multiprocessor Operating System Organizations 15
1.6.3 Program Instrumentation 16
1.7 Conclusions 17
1.8 Future Work 18

2 Paper I **21**
2.1 Introduction 21
2.2 Multiprocessor Port Challenges 22
2.3 Categorization 23
2.3.1 Giant Locking 23
2.3.2 Coarse-grained Locking 24
2.3.3 Fine-grained Locking 25
2.3.4 Lock-free Approaches 26
2.3.5 Asymmetric Approaches 26
2.3.6 Virtualization 26

2.3.7 Reimplementation 27

2.4 Operating System Implementations 27

2.4.1 Locking-based Implementations 27

2.4.2 Lock-free Implementations 29

2.4.3 Asymmetric Implementations 30

2.4.4 Virtualization 31

2.4.5 Reimplementation 31

2.5 Linux Case Study 32

2.5.1 Evolution of Locking in Linux 32

2.5.2 Locking and Source Code Changes 33

2.5.3 Performance Evaluation 34

2.6 Discussion and Conclusions 36

3 Paper II **39**

3.1 Introduction 39

3.2 The Operating System 41

3.2.1 The Programming Model 42

3.2.2 The Distributed Main-Memory Database 42

3.2.3 The Process and Memory Model 43

3.3 Design of the Multiprocessor Support 45

3.3.1 Kernel Locking and Scheduling 46

3.3.2 CPU-local Data 46

3.3.3 Multithreaded Processes 47
3.4 Evaluation Framework 50
3.5 Evaluation Results 51
3.6 Implementation Experiences 52
3.7 Related and Future Work 53
3.8 Conclusions 56

4 Paper III **57**
4.1 Introduction 57
4.2 Related Work 59
4.2.1 Monolithic Kernels 60
4.2.2 Microkernel-based Systems 61
4.2.3 Asymmetric Operating Systems 61
4.2.4 Cluster-based Approaches 63
4.3 The Application Kernel Approach 63
4.3.1 Terminology and Assumptions 64
4.3.2 Overview 64
4.3.3 Hardware and Software Requirements 66
4.3.4 Application Kernel Interaction 67
4.3.5 Exported Application Programming Interface 69
4.4 Implementation 70
4.4.1 Paging 72

4.4.2 `clone`/`fork` System Calls . . . 73
4.4.3 Running Applications . . . 75
4.5 Experimental Setup and Methodology . . . 76
4.5.1 Evaluation Environment . . . 76
4.5.2 Benchmarks . . . 77
4.6 Experimental Results . . . 79
4.6.1 Performance Evaluation . . . 79
4.6.2 Implementation Complexity and Size . . . 81
4.7 Conclusions and Future Work . . . 83

5 Paper IV **87**

5.1 Introduction . . . 87
5.2 Background . . . 89
5.2.1 Instrumentation approaches . . . 89
5.2.2 Instrumentation perturbation . . . 91
5.3 The LOPI instrumentation framework . . . 93
5.4 Measurement methodology . . . 99
5.5 Measurement results . . . 101
5.6 Related work . . . 106
5.7 Conclusions . . . 108

List of Figures

1.1 Development effort and performance tradeoff. 17

2.1 Multiprocessor operating system oganizations. 24

2.2 Results from the postmark benchmark 36

2.3 Design space for multiprocessor implementations 37

3.1 In-house operating system structure 41

3.2 IA-32 address space layout 44

3.3 Container handling 45

3.4 Handling of multithreaded containers 48

4.1 The application kernel approach 65

4.2 Trap handling in the application kernel 68

4.3 Application kernel shared area layout 72

4.4 Application kernel device driver structure 73

4.5 Application kernel clone handling 74
4.6 Application startup 75
4.7 Parallel and multiprogramming benchmark results 82
4.8 McCabe complexity 84

5.1 The instrumentation process 91
5.2 A non-instrumented function call. 93
5.3 An instrumented function call 94
5.4 An instrumented function return 94
5.5 Pseudo code for the `instr_func_enter`-function. 95
5.6 Pseudo code for the `instr_func_leave`-function. 98
5.7 Cycles per function call 100
5.8 Execution profile for two SPEC benchmarks. 104

List of Tables

2.1 Categorized multiprocessor operating systems. 28

2.2 Locks in the Linux kernel 34

2.3 Code lines in multiprocessor Linux 35

3.1 User/kernel execution time 51

4.1 Application kernel benchmarks 78

4.2 Application kernel getpid latency 80

4.3 Single-process benchmark results 80

4.4 Parallel and multiprogramming benchmark results 81

4.5 Comment-free lines of code 83

5.1 Benchmark description 99

5.2 SPEC benchmark overhead. 102

Chapter 1

Introduction

1.1 Background

A current trend in the computer industry is replacing uniprocessor computers with small multiprocessors. Traditionally, most small multiprocessors have been SMPs (Symmetric Multiprocessors) with two or more processor chips where each processor has equal access to memory and hardware devices. This situation is changing, however. Currently, more and more manufacturers are starting to produce processors where the chip itself is able to execute more than one program or program thread at a time.

The trend of parallelized processors has two main tracks. First, symmetric multithreading (SMT) [26] is an approach where processor resources are shared between two or more concurrently running program threads to allow more efficient use of processor resources. Symmetric multithreading has been implemented in, e.g., current Intel Pentium 4 and Xeon processors [67] (branded as HyperThreading), and the 2-way multithreaded Sony/IBM Cell processor [83].

The second track is chip multiprocessors (CMPs) [37]. This approach partitions the chip area into two or more mostly independent processor cores. This means that in the CMP case, the processor resources are statically allocated to a core whereas the resources in an SMT are dynamically allocated to a processor thread. The argument for chip multiprocessors is similar to that for simultaneous multithreading: using many simple cores provides better energy efficiency and higher operating frequency than using one complex core. Chip multiprocessors have recently been announced by Intel [42] and AMD [2], and the IBM POWER4 architecture was released as a dual-core chip in 2001 [46].

Future microprocessors will in many cases contain elements of both the SMT and CMP approaches. For example, the IBM POWER5 [47] architecture also employ a combination of CMP and SMT technologies. Further, the new Intel chip multiprocessors (Intel Pentium processor Extreme Edition) also supports HyperThreading. As a result of these trends, a lot of future microprocessors will be small multiprocessors, and multiprocessors will thus be used in a large number of applications.

Developing programs for multiprocessors presents special problems. While program threading is possible on uniprocessor computers, truly concurrent program threads are only available on a multiprocessor. True concurrency can expose problems which never occurs on a uniprocessor, e.g., concurrent update of a shared variable. It also makes debugging more difficult since controlling other threads of execution is harder than on a uniprocessor.

Operating systems need to be adapted to work with the new multiprocessors. It is not possible to simply allow several processors to start executing in a uniprocessor operating system, since this would cause unpredictable behavior when processors concurrently modifies a data structure. Instead, mutual exclusion is needed in multiprocessor operating systems, e.g., through locking, so that a processor is stalled or redirected when trying to access a resource held by another processor. For good performance, it is important that the mutual exclusion is fine-grained enough to avoid spending time waiting for other processors.

However, making modifications to large software bodies such as operating system kernels is generally very hard. For instance, Freeman L. Rawson III writes about this problem when developing the IBM Workplace OS [85], which was canceled after six years of development by a large team of engineers because of engineering problems and performance issues. In addition, operating system kernels are in many cases harder to develop and debug than other software systems because of non-predictability, limited library support, real-time issues, and hardware effects.

There are performance issues associated with applications running in multiprocessor operating systems. First, a non-threaded application will not benefit from a multiprocessor. Second, applications can be categorized into two broad classes, *compute bound* and *system bound*, where compute bound applications spend most of their time executing application code and system bound applications invoke the operating system frequently. Compute-bound applications are not dependent on the parallelization of the operating system and can therefore easier benefit from multiprocessors even on modestly parallelized operating systems. Third, there are theoretical limits to the performance obtainable from a multiprocessor. Amdahl's law [3] states that since all programs need to execute parts serialized (system calls, printouts, etc.), the highest speedup attainable with a given number of processors is:

$$speedup = \frac{1}{t_{serial} + \frac{t_{parallel}}{\#processors}}$$

With Amdahl's law, the maximum speedup theoretically possible is given by setting the processors to ∞. For example, with an application that spends 10% of the time running serialized code, the maximum speedup is $\frac{1}{0.1+\frac{0.9}{\infty}} \approx 10$, i.e., even a system with thousands of processors could never achieve more than a speedup of 10 compared to a uniprocessor.

Porting an operating system to a multiprocessor computer with good performance characteristics can therefore demand a tremendous effort, which can pose problems even to large organizations and experienced software developers. In this thesis, the tradeoff between performance and

implementation effort for operating system kernels is therefore explored, focusing on techniques for reducing the development effort.

The rest of the introduction is organized as follows. In Section 1.2 the research questions posed in this thesis are outlined. Thereafter, the main contributions from the thesis are described in Section 1.4. In Section 1.3, the research methodology used is discussed. Section 1.7 presents the conclusions of this thesis and Section 1.8 outlines future work.

1.2 Research Questions

The work in this thesis is centered around the implementation of multiprocessor support in operating system kernels, specifically looking at ports of existing uniprocessor kernels. In my research, I have tried to find a balance between development effort and performance when implementing the multiprocessor support. The primary research question is therefore:

- **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?

The primary research question is discussed directly in Paper I, while the other papers deal with specialized cases of it. The primary research question has therefore been further subdivided into three more specific questions. The initial aim was to explore the effort and performance of traditional multiprocessor porting methods, so the first specific question is:

- **RQ1, traditional**: What is the cost and performance benefit of performing a traditional symmetric lock-based multiprocessor port of an operating system kernel?

This question is discussed in Paper I and Paper II. Paper I contains an overview of different multiprocessor porting approaches, and Paper II describes the implementation of a giant locking scheme in an operating system kernel. Having discussed the traditional approaches, the next goal was to explore the development effort needed to port an operating system. The next question relates to these alternative organizations:

- **RQ2, alternative organizations**: 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?

The second research question is answered in Paper I and Paper III. Finally, in large and complex software projects such as operating system kernels, support tools for development often play a crucial role. One particularly important area is program instrumentation, i.e., adding probes to the program to detect bugs or performance problems. Especially in the context of performance instrumentation, it is important that the instrumentation perturbs the original program as little as possible. The final research question, which is dealt with in Paper IV, is therefore:

- **RQ3, support tools**: How is it possible to lower the perturbation caused by program instrumentation?

1.3 Research Methodology

1.3.1 Research Methods

In this thesis, there are two basic issues: performance and development effort. To address these issues, a quantitative benchmark-based approach has been used. In our studies, a combination of application benchmarks and synthetic benchmarks has been used for the performance evaluations. Theoretical analysis, which serves to establish performance limits, has also

been employed in some cases. The benchmarks have been run either in the Simics full system simulator [65] or on real hardware. When evaluating development effort, time has been used as the premier attribute, but there are also measurements of code properties such as number of code lines and McCabe cyclomatic complexity [27].

Performance Evaluation

Generally, scripted application benchmarks are used to measure the performance of a real application with specified input and output [39, page 27]. Synthetic benchmarks (or micro benchmarks), on the other hand, measure specific parts of an execution such as the time required for a system call.

For the evaluations, a combination of the standard SPEC CPU 2000 [94], SPLASH 2 [106], more specialized benchmarks such as Postmark [49], and custom synthetic benchmarks has been used. SPEC CPU 2000 contains a set of mainly compute-bound single-threaded benchmarks which is often used in computer architecture research, compiler research, and computer systems performance evaluation. SPLASH 2 is a benchmark suite with parallel scientific applications commonly used in evaluating multiprocessor performance. As the SPEC applications are single-threaded, they will not benefit from running on a multiprocessor machine unless run in a multiprogramming setting. Most of the SPLASH benchmarks, on the other hand, scale well on multiprocessor machines.

The MinneSPEC reduced workloads [51] has been used to decrease the simulation time of SPEC CPU 2000. The MinneSPEC workloads are constructed to have similar characteristics as the full workloads, although recent work have shown that periodic sampling gives a closer correspondence to the full workloads [108].

Since both SPEC and SPLASH are compute-bound, the operating system will have only minor influence on the performance. In contrast, the Postmark [49] benchmark is a mainly system bound benchmark. Postmark models the behavior of a mail server, focusing on the performance of

many small files. Since file system operations to a large extent are handled in-kernel, Postmark will spend significant time executing kernel code. The parallelization of the operating system kernel is therefore very important when running the Postmark benchmark, and only highly parallel operating system kernels will scale well with this benchmark.

In some cases it has not been possible to use standard benchmarks. For example, in Paper III we measure the latency of a "null" system call, and for this we use a custom synthetic benchmark measuring that particular operation. Further, in Paper II, it was not possible to use standard benchmarks because of the specialized platform, and configuration problems also prohibited the use of normal applications on the operating system. Instead, we constructed a custom application with one thread per processor that makes it possible to measure performance during varying system call load, showing how the parallelization of the kernel affects performance.

In Paper IV, we use application benchmarks to compare different instrumentation techniques. The application benchmarks measure the aggregate behavior of the SPEC applications during the entire run. We measure the number of instructions executed, cache accesses and misses as well as branches and branch prediction misses. The measurements were performed on real hardware.

Development Effort Measurement

To measure development effort, McCabe cyclomatic complexity has been used. McCabe cyclomatic complexity measures the possible number of paths through a function, which is a measure of how complex the function is to understand and analyze. Generally, the fewer paths through a function, the fewer test cases are needed to test the function [27]. The McCabe cyclomatic complexity has been measured with the `Pmccabe` [10] tool.

The number of source code lines is also an important characteristics for the development effort. Except where otherwise stated, the the `sloc-`

`count` [104] tool by David A. Wheeler, which counts comment-free lines of source code in many languages, has been used.

1.4 Contributions in this Thesis

In this section, the papers in the thesis are discussed together with the contributions made in each paper.

1.4.1 Chapter 2 (Paper I)

Chapter 2 presents an investigation of scalability and development effort for operating system kernels. The purpose of this paper is to explore the tradeoff between these two quality attributes in the context of operating system kernels, specifically when porting a uniprocessor kernel to multiprocessor hardware. In this paper, we identify seven technical approaches for performing a multiprocessor port and discuss these in the context of performance and development effort. Further, we perform a case study of how Linux multiprocessor support has evolved in terms of performance and implementation complexity for four different versions of the Linux kernel.

The main contribution of this paper is the categorization of technical approaches for operating system kernels and also a discussion of the scalability and development effort for these. In the paper, we argue that the technical approach has a significant effect on the development effort, ranging from approaches with very low effort such as the approach we present in Paper III to complete reimplementations with very high implementation complexity. In the same way, the achieved performance will vary according to the chosen approach, and generally the expected pattern of higher development effort correlating with higher scalability holds. We base the results on a substantial literature study and a case study of the Linux operating system.

This paper connects directly to the main research question, regarding the performance and development effort tradeoff in multiprocessor operating system ports, which we try to answer for different technical approaches to operating system porting. Further, it also relates to RQ1 and RQ2 on a higher level, since it gives an overview of both traditional and alternative systems. The paper is also a foundation for the later papers, which discuss the more specific research questions.

1.4.2 Chapter 3 (Paper II)

In Chapter 3, we present the design and implementation of multiprocessor support for a large industrial operating system kernel using a traditional porting approach. The main purpose of this paper is to discuss the issues and design options when porting a full-scale operating system kernel to multiprocessor hardware. In the paper, we present implementation details, an initial evaluation of the implementation, and experiences we gained from the implementation.

There are two main contributions in this paper. First, we illustrate technical solutions to common problems found in multiprocessor ports. For example, we use an approach for processor-local data based on virtual memory mapping which makes it possible to keep most of the uniprocessor code unchanged. This approach has a few problems related to multithreaded processes, and the paper presents a solution to these problems with minimal changes to the original kernel. The second contribution is a discussion of the experiences we gained from the implementation. Although we chose a simple approach, the implementation still required around two years, which was more time than we had expected. The main reason for this is the large code-base, which meant that we had to spend a lot of time understanding the structure and implementation of the system. Further, the system is highly specialized and employs a complex configuration process which required us to spend time on getting the environment to work.

With this paper, we discuss the *traditional* research question (RQ1). The paper provides implementation experiences from a traditional port,

which was harder to implemented than we had expected. The paper also connects to the main research question.

1.4.3 Chapter 4 (Paper III)

The traditional approach we used in Paper II allowed us to successfully port a large industrial operating system kernel, but at the cost of long implementation time. Chapter 4 presents an alternative approach, the *application kernel approach*. The application kernel approach provides a way of adding multiprocessor support without changing the original uniprocessor kernel. The approach achieves this by running two kernels in parallel, the original uniprocessor kernel on one processor and a custom kernel on all other processors. The paper describes an implementation of the application kernel for Linux, which shows that the approach is feasible in a real-world setting. We need no changes to neither the Linux kernel nor the applications.

The main contribution from Paper III is that we show that it is possible and feasible to add multiprocessor support to an operating system with minimal changes to the original operating system. We also evaluate the application kernel approach in terms of performance and implementation complexity, where we show that the application kernel is comparable in performance to Linux for compute-bound applications. We therefore conclude that the application kernel approach would be a viable alternative for multiprocessor ports of complex operating systems focusing on computationally intensive applications.

Paper III discusses the research question about *alternative organizations*, showing that alternative organizations can provide significant advantages in terms of implementation effort. It also connects directly to the main research question, focusing on an approach with low implementation complexity.

1.4.4 Chapter 5 (Paper IV)

The last paper describes the LOPI framework for program instrumentation. LOPI is a generic framework for low overhead instrumentation of program binaries. LOPI allows arbitrary instrumentation to be added to the program binary, e.g., performance measurements or path profiling. In LOPI, we provide a number of low-level optimizations to reduce the perturbation caused by the instrumentation framework. For example, LOPI tries to improve cache locality by reusing instrumentation code whenever possible. With these optimizations, LOPI is able to perform significantly better than the state-of-the-art Dyninst [16] instrumentation package.

The main contribution of Chapter 5 is that we show how a number of low-level optimizations can be used to improve program instrumentation perturbation. LOPI also provides the possibility of automatically adding tests or performance measurements to large software packages without changing the source code, or even having access to the source code. We believe that optimizations such as these can improve the accuracy of measurements and experiments performed using instrumentation.

In this paper, we discuss the research question about *support tools*. This also connects to the main research question in that support tools are vital components when working on large software packages such as operating systems.

1.5 Validity of the Results

This section presents a number of threats to the validity of the studies. The discussion is centered around threats to the generalizability (or external validity) and the internal validity.

1.5.1 Generalizability, External Validity

External validity refers to the possibility of generalizing the study results in a setting outside of the actual study [87, page 106]. There are two threats to the generalizability of the work which are applicable to all papers in this thesis. One dimension is hardware, i.e., if the results are generalizable to larger hardware configurations or portable to entirely different platforms. The other dimension is software, i.e., if the results are generalizable to other software platforms.

The hardware generalizability is addressed in several ways. First, Paper IV describes low-level optimizations which are closely tied to the target hardware and therefore hard to port and generalize. In this case, this is inherent in the approach since the optimizations are intentionally system dependent. Paper II and Paper III describe multiprocessor ports of operating system kernels, both targeting Intel IA-32. While many low-level aspects of these ports are architecture-specific, e.g., startup of secondary processors, most of the code (locking, etc.) is common between architectures which makes most of the results valid for other architectures as well. The application kernel approach presented in Paper III poses a set of requirements on the architecture, e.g., processor-local interrupt handlers. These requirements are discussed for different architectures, and in most cases these are trivially fulfilled. It is therefore likely that the application kernel can be easily ported to a large set of hardware platforms.

Hardware scalability threats has been further addressed by using the Simics full-system simulator [65] to simulate hardware configurations with between one and eight processors. Simics gives the possibility to test larger configurations than the available hardware, and is used to examine the scalability in Paper I and Paper III. Further, Simics has been used in Paper IV to study detailed application behavior, which is hard or impossible using traditional hardware measurements.

To improve the software generalizability, standard benchmarks such as SPEC CPU 2000, SPLASH 2, and Postmark have been employed in Paper I, Paper III and Paper IV. Using standard benchmarks allow the studies to be replicated and also to be compared with other similar studies.

It is also shown that the application kernel approach is fairly independent of the original uniprocessor kernel, since most of the code from an in-house kernel implementation could be reused for the Linux port of the application kernel. This suggests that the application kernel approach should be possible to reuse mostly unmodified for ports to other operating systems.

The specialized benchmark used in Paper II is harder to generalize. Still, since the benchmark is very basic, measuring the parallelization of the operating system kernel, it is still possible to compare to similar benchmarks on other operating systems. However, since it was not possible to run full-scale applications, it is difficult to generalize the performance results to a production environment.

1.5.2 Internal Validity

Internal validity refers to how well a study can establish the relationship between cause and effect [87, page 103]. For example, in a performance comparison between two versions of a multiprocessor operating system kernel (which was done for the Linux kernel in Paper I), a finding might be that the later version of the kernel has better performance. However, it is not possible to draw the conclusion that this is because of improved multiprocessor support in the newer version since other factors such as a better file system implementation, optimized virtual memory handling, etc., also affects performance. In Paper I, the Linux kernel performance results is therefore normalized against a baseline of uniprocessor performance in the 2.0 kernel. The benchmark also shows that the performance, even on the uniprocessor, is around three times higher on 2.6 than on 2.0 for the same benchmark.

One validity concern for the work is the use of a full-system simulator compared to real hardware. If the simulator is not accurate enough, the results will diverge from real hardware. There are two aspects of this. First, the Intel IA-32 instruction set is very complex to simulate accurately, compared to many RISC architectures. On current processors, IA-32 instructions are split up in RISC-like micro instructions [44] before

they are executed, and the actual microcode implementation can vary greatly between different implementations of the architecture. Although Simics simulates microcode for the Pentium 4, the available hardware at the time of the experiments (Pentium Pro, Pentium II, and Pentium III) is quite different from Pentium 4 and microcode simulation was therefore not employed.

The second aspect is cache simulation. Since main memory is several magnitudes slower than the processor, caches that store frequently used data are needed to hide memory latency [39]. A miss in the cache can stall the processor for significant durations. For multiprocessors, another source of latency is traffic to keep the cache contents on different processors coherent. Simics can simulate the complete memory hierarchy including caches and coherence traffic. In Paper IV, memory hierarchy simulation has been used to show detailed behavior of the instrumentation of SPEC applications (Section 5.5 in Chapter 5), whereas measurements on real hardware were used to get aggregate values.

In Paper III, the memory hierarchy was not simulated since the purpose of the simulations have been to show scalability limits. There are known performance issues with the application kernel prototype related, e.g., to memory layout, which are not implemented and would give a disproportionally large disadvantage in the cache simulation.

1.6 Related Work

The related work has been divided after the subsidiary research questions. Since Paper I contains both traditional multiprocessor ports and alternative organizations, this paper is discussed in the context of the corresponding subsidiary questions.

1.6.1 Traditional Multiprocessor Ports

The traditional giant locking organization used in Paper II has been used in a number of other systems. For example, early versions of Linux and FreeBSD both used a giant locking approach [13, 60]. As discussed in Paper I, the giant locking approach is a feasible approach for initial multiprocessor ports since it is relatively straightforward to implement and also possible to incrementally enhance by making the locks more fine-grained.

The system used in Paper II is a cluster operating system kernel running on IA-32 hardware, and there exists other similar systems. While generic operating system kernels such as Linux [15], Windows NT [71] and Sun Solaris [97] have been used to build cluster systems, there are fewer dedicated operating system kernels for clusters. One example of such a system is Plurix [33]. Plurix is a kernel which, like the system presented in Paper II, employs distributed objects which are kept coherent through a transaction-based scheme. However, Plurix only runs on uniprocessor nodes and is also based on Java, whereas the multiprocessor port in Paper II supports both Java and C++ development.

1.6.2 Alternative Multiprocessor Operating System Organizations

The work in Paper III has several connections to other work. First, there is other work done related to employing restricted knowledge in systems. For example, Arpaci-Dusseau et al. [7] propose a method where "gray-box" knowledge about algorithms and the behavior of an operating system is used to acquire control and information about the system without explicit interfaces or operating system modification. This idea is similar to Paper III in that it restricts the information needed about the kernel to knowledge about the algorithms, but it differs in the intended purpose: controlling operating system behavior compared to adding multiprocessor support. Zhang et al. [109], have done work where the operating system kernel is modified to provide quality of service guarantees to large unmodified applications. This work takes the opposite approach to Paper III:

the kernel is explicitly modified to suit applications while the approach in Paper III avoids modifying the kernel and actually needs no modifications to the applications either.

Second, the technical approach described in Paper III is related to older operating system kernels organized as master-slave systems and to certain distributed systems. For example, Goble and Marsh [32] describe the hardware and software implementation of a master-slave VAX multiprocessor. Paper III uses the same structure as master-slave systems in that it restricts kernel operations to one processor. However, master-slave systems modify the original kernel to add the multiprocessor support whereas the application kernel approach adds the support outside of the original kernel. Further, the MOSIX distributed system [11], which provides support for distributing standard applications transparently on a cluster, also uses a similar approach and redirects kernel operations to the "unique home node" of an application. The approach in Paper III works the same way, but on a multiprocessor computer instead of a cluster.

1.6.3 Program Instrumentation

There are several instrumentation tools which share properties with LOPI. First, there are a number of tools which directly employ binary rewriting similar to LOPI. For example, Etch [88] is a tool for rewriting Windows binaries on the IA-32 architecture, which like LOPI has to deal with the complexities of an instruction set with variable-sized instructions. EEL [58] and ATOM [4] also rewrite binaries, but have been constructed for the SPARC and Alpha architectures, respectively, which use fixed-sized instructions and are therefore better adapted to instrumentation. Both EEL and ATOM provide frameworks to build tools to instrument programs, unlike LOPI which only provides basic instrumentation since the purpose of LOPI is to provide low-level support for building instrumentation tools.

Dyninst [16] and Pin [64] use a different approach than LOPI for the instrumentation. Both these tools allow programs to be dynamically instrumented, i.e., adding instrumentation to the program in-memory after the program has been started. Although LOPI only instruments binaries,

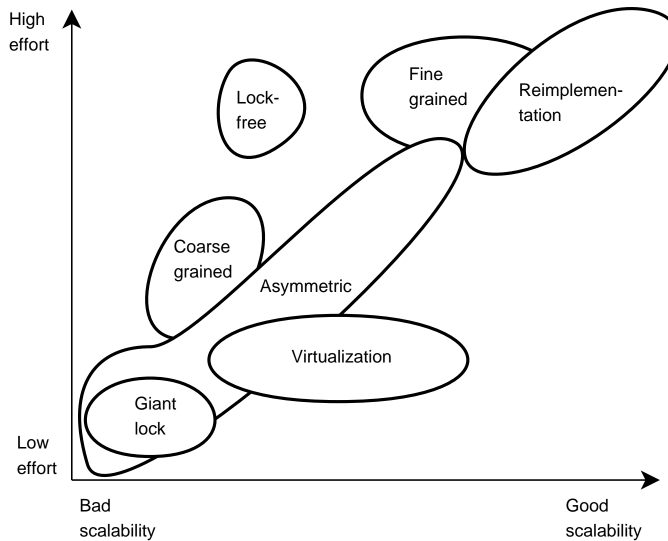


Figure 1.1: The tradeoff between development effort and performance for different approaches to multiprocessor operating system kernels.

the optimizations performed by LOPI is applicable to dynamic instrumentation as well.

1.7 Conclusions

The primary goal of this thesis has been to look into the tradeoff between performance and development effort for the implementation of multiprocessor operating system kernels. I have investigated this tradeoff both for traditional methods of porting uniprocessor kernels in Paper I and Paper II and for alternative organizations in Paper I and Paper III. Although not strictly tied to multiprocessor operating systems, Paper IV presents a tool for program instrumentation which can be used during the development of large software systems such as operating system kernels.

In these studies, I found that there exists a large diversity of approaches with different characteristics in terms of performance and development

effort. Figure 1.1 shows an approximation of the design space for different multiprocessor porting approaches, taken from Paper I, and this figure summarizes the findings of this thesis. In Paper II, a traditional giant locking approach was employed for the multiprocessor port, which was possible to implement with relatively small changes to the original kernel, but also has limited performance. In this case, the port still took a lot of time due to a complex code base and configuration problems.

Alternative approaches such as the application kernel approach described in Paper III could substantially reduce the cost of initially porting an operating system to a multiprocessor. Most of the application kernel code is operating system independent, and further only requires basic hardware support (processor startup and timers). Therefore, the application kernel approach should be easy to port to other kernels. The application kernel approach could for example be used to quickly get a first working multiprocessor port fast, while a more targeted method is being developed. It also presents other advantages, e.g., automatic propagation of improvements of the uniprocessor kernel to the multiprocessor port, and there is further no need to keep two versions of the code base.

1.8 Future Work

There are several directions in which future work could proceed. First, an investigation of the generalizability of the application kernel approach could be made. Such an investigation would focus on separating the architecture-dependent and operating system-dependent parts of the application kernel. The vision is to create the possibility of drop-in addition of multiprocessor support, with the implementer only providing minimal wrappers for the new operating system.

Second, there are a number of optimizations that can be performed for the application kernel approach. For example, one possibility is to look into hybrid approaches where the application kernel itself handles certain classes of operations, e.g., simpler system calls such as returning the process id. Another idea is to monitor the applications to dynamically

move system-bound applications to execute on the original uniprocessor kernel directly and conversely move compute-bound applications to the application kernel.

Third, the multiprocessor port in Paper II could be improved in a number of ways. An obvious improvement is to relax the giant locking scheme to support more coarse-grained locking, starting with a separate lock for low-level interrupt handling. This is likely to improve the performance of system bound processes, but also makes the implementation more complex. It is also possible to further examine where the uniprocessor kernel spends its time to see which areas to target with subsystem locks.

Fourth, we would also like to investigate alternatives to the giant locking approach we implemented in Paper II. Primarily, we would like to look into virtualization techniques, e.g., using Xen [12], where a large multiprocessor computer is partitioned into several independent cluster nodes. There are problems specific to the virtualization option such as database replication (the replication must not be done between two nodes on the same machine) which we aim to explore. The virtualization approach could then be compared to the current multiprocessor port in terms of implementation complexity and performance.

A fifth possibility is to look deeper into support tools for the development of operating system kernels. Debugging operating system kernels is quite hard because of real-time issues, hardware interaction and lack of support for standard debuggers. One way of easing the debugging process is by use of a full-system simulator such as Qemu [14] or Simics, which has been used in debugging the Linux kernel [1]. More recently, Simics has also gained the possibility to "go back in time", i.e., to step backwards while debugging [103]. However, while full-system simulators provide a very valuable framework for debugging, they are not always very easy to use. For example, debuggers usually assume that only one process is debugged, which is true when running on real hardware, whereas a full-system simulator has no concept of process. Therefore, a possible area of future work is to investigate how debuggers can be extended to support the possibilities offered by full-system simulators.

Chapter 2

Paper I

Scalability vs. Development Effort for Multiprocessor Operating System Kernels

Simon Kågström, Håkan Grahn, Lars Lundberg

Submitted for journal publication, August 2005

2.1 Introduction

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

There are a number of possible technical approaches when porting an operating system to a multiprocessor, e.g., introducing coarse or fine

grained locking of shared resources in the kernel, introducing a virtualization layer between the operating system and the hardware, using master-slave or other asymmetric solutions.

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

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

The rest of the paper is structured as follows. Section 2.2 describes the challenges faced in a multiprocessor port. In Section 2.3 we present a categorization of porting methods, and Section 2.4 then categorizes a number of existing multiprocessor systems. In Section 2.5, we describe our Linux case study, and finally discuss our findings and conclude in Section 2.6.

2.2 Multiprocessor Port Challenges

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

single virtual page and map this page to different physical memory pages for every CPU.

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

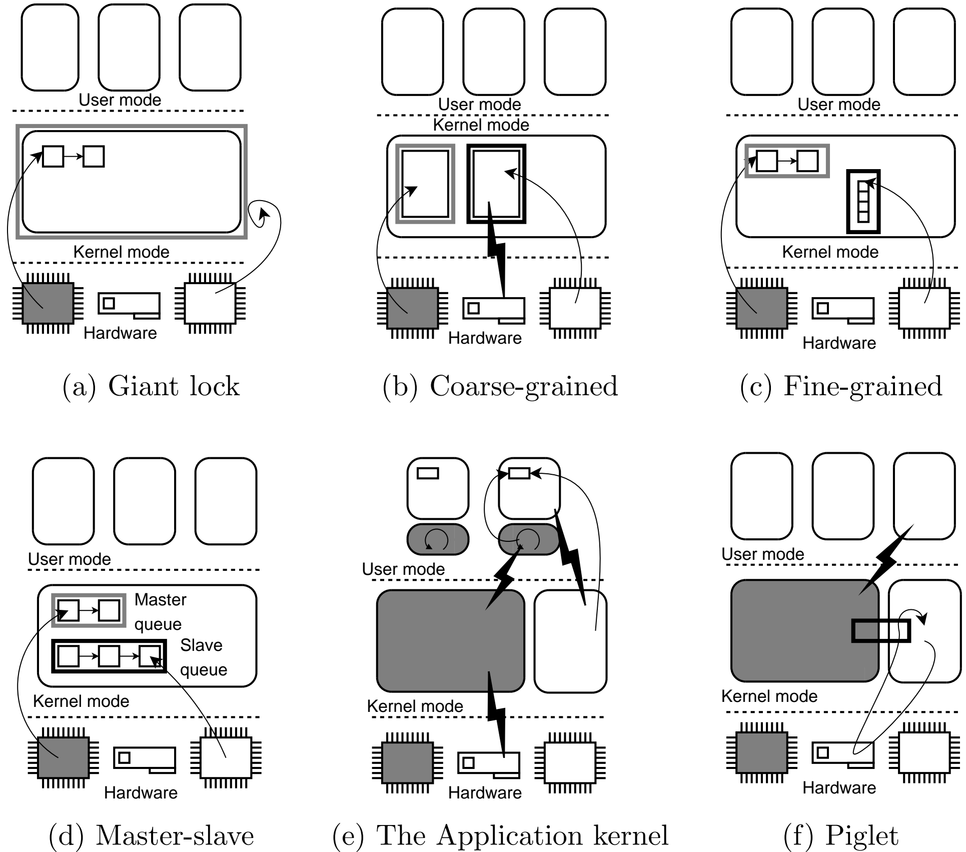
Finally, since many modern processors employ memory access reordering to improve performance, memory barriers are sometimes needed to prevent inconsistencies. For example, a structure need to be written into memory before the structure is inserted into a list for all processors to see the updated data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

2.3 Categorization

We have categorized the porting approaches into the following: *giant locking*, *coarse-grained locking*, *fine-grained locking*, *lock-free*, *asymmetric*, *virtualization*, and *reimplementation*. Other surveys and books [77, 91] use other categorizations, e.g., depending on the structuring approach (microkernels and monolithic kernels). Figure 2.1 illustrates the different methods as well as a few specific implementations (described in Section 2.4).

2.3.1 Giant Locking

With giant locking (Figure 2.1a), a single spin lock protects the entire kernel from concurrent access. The giant lock serializes all kernel accesses, so most of the uniprocessor semantics can be kept. Giant locking requires small changes to the kernel apart form acquiring and releasing the lock, e.g., processor-local pointers to the current process. Performance-wise, scalability is limited by having only one processor executing in the kernel at a time.



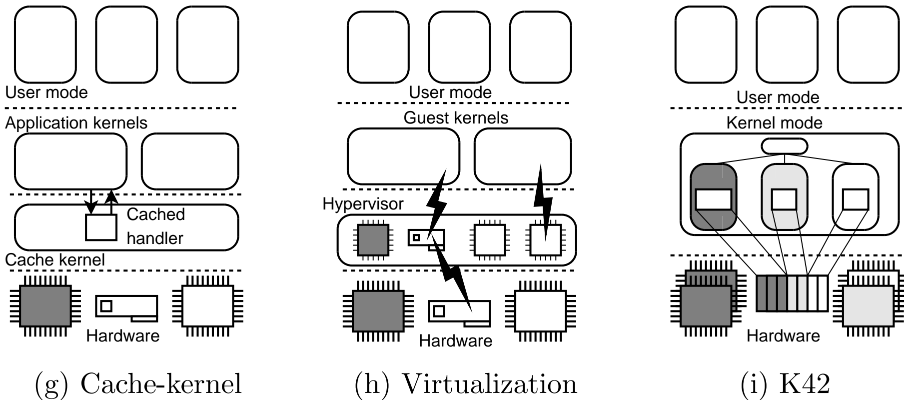
(a) Giant lock (b) Coarse-grained (c) Fine-grained

(d) Master-slave (e) The Application kernel (f) Piglet

Figure 2.1: *Continued on next page*

2.3.2 Coarse-grained Locking

Coarse-grained locks protect larger collections of data or code, such as entire subsystems as shown in Figure 2.1b. Compared to giant-locking, coarse-grained locks open up for some parallel work in the kernel. For example, a coarse-grained kernel can have separate locks for the filesystem and network subsystems, allowing two processors to execute in different subsystems. However, inter-dependencies between the subsystems can force an effective serialization similar to giant locking. If subsystems are



(g) Cache-kernel (h) Virtualization (i) K42

Figure 2.1: Multiprocessor operating system organizations. Thick lines show locks and the flash symbol denote system calls or device interrupts. The figure shows both categories and examples of systems.

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

A special case of coarse-grained locking is *funnels*, used in DEC OSF/1 [23], and Mac OS X [31]. In OSF/1, code running inside a funnel always executes serialized on a master processor, similar to master-slave systems. Mac OS X does not restrict the funnel to a single processor. Instead the funnel acts as a subsystem lock, which is released on thread rescheduling.

2.3.3 Fine-grained Locking

Fine-grained locking (Figure 2.1c), restricts the locking to individual data structures or even parts of data structures. Fine-grained locking allows for increased parallelism at the cost of more lock invocations and more complex engineering. Even fine-grained implementations will sometimes use coarse-grained locks, which are more beneficial for uncontended data.

2.3.4 Lock-free Approaches

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

A completely lock-free operating system relies on these specialized data structures. Lock-free data structures are sometimes hard to get correct and can be inefficient without proper hardware support [25], which limits the scalability of completely lock-free approaches. Also, making an existing kernel lock-free requires a major refactoring of the kernel internals, so the development costs of a lock-free kernel is likely to be high.

2.3.5 Asymmetric Approaches

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

2.3.6 Virtualization

A completely different method is to partition the multiprocessor machine into a virtual cluster, running many OS instances on shared hardware (Figure 2.1h). This category can be further subdivided into fully virtualized and paravirtualized systems, where the latter employs operating

system modifications and virtual extensions to the architecture to handle hardware deficiencies or lower virtualization overhead.

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

2.3.7 Reimplementation

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

2.4 Operating System Implementations

In this section, we discuss different implementations of multiprocessor ports. Table 2.1 provides a summary of the discussed systems.

2.4.1 Locking-based Implementations

Many to multiprocessor ports of uniprocessor operating systems are first implemented with a giant lock approach, e.g., Linux 2.0 [13], FreeBSD [60], and other kernels [53], and later relaxes the locking scheme with a more fine-grained approach. The QNX Neutrino microkernel [84], also protect the kernel with a giant lock. However, as most operating system functionality is handled by server processes, the parallelization of these is more important than the actual kernel.

Mac OS X started out with what was effectively a giant lock (a funnel for the entire BSD portion of the kernel), but thereafter evolved into a more

Table 2.1: *Continued on next page.*

System	Method	Focus
Linux 2.0 [13]	Giant	General purpose
FreeBSD 4.9 [60]	Giant	General purpose
QNX [84]	Giant	Real-time
Linux 2.2	Coarse	General purpose
Mac OS X [31]	Coarse	General purpose
OSF/1 [23]	Fine	General purpose
Linux 2.4	Fine	General purpose
Linux 2.6 [63]	Fine	General purpose
AIX [22, 99]	Fine	General purpose
Solaris [50]	Fine	General purpose
FreeBSD 5.4	Fine	General purpose
Synthesis [68]	Lock-free	General purpose
Cache kernel [21]	Lock-free	Application specific
Dual VAX 11/780 [32]	Asymmetric	General purpose
Application kernel [57]	Asymmetric	Low effort
Piglet [76]	Asymmetric	I/O intensive
Cellular Disco [34]	Virtualized	Hardware sharing, fault tolerance
VMWare ESX [90]	Virtualized	Hardware sharing
L4Linux [102]	Virtualized	Hardware sharing
Adeos [107]	Virtualized	Hardware sharing
Xen [12]	Virtualized	Hardware sharing
K42 [6]	Reimpl.	Scalability

coarse-grained implementation with separate funnels for the filesystem and network subsystems. Currently, Mac OS X is reworked to support locking at a finer granularity.

AIX [22] and DEC OSF/1 3.0 [23] were released with fine-grained locking from the start. In both cases, the SMP port was based on a preemptible uniprocessor kernel, which simplified porting since disabling of preemption correspond to places where a lock is needed in the multiprocessor version. During the development of OSF/1, funneling was used to protect the different subsystems while core parts like the scheduler and virtual memory were parallelized. Solaris [50] and current versions of Linux [63] and FreeBSD also implement fine-grained locking.

Table 2.1: Summary of the categorized multiprocessor operating systems. The code lines refer to the latest version available and the development time is the time between the two last major releases.

	Performance		Effort	
System	Latency	Scalability	Code lines	Devel. time
Linux 2.0 [13]	High	Low	955K	12 months
FreeBSD 4.9 [60]	High	Low	1.9M	?
QNX [84]	Low	?	?	?
Linux 2.2	Medium	Low	2.5M	18 months
Mac OS X [31]	High	Low	?	?
OSF/1 [23]	Low	High	?	?
Linux 2.4	Medium	Medium	5.2M	11 months
Linux 2.6 [63]	Low	High	6.1M	11 months
AIX [22, 99]	Low	High	?	18 months
Solaris [50]	Low	High	?	?
FreeBSD 5.4	Low	High	2.4M	?
Synthesis [68]	Low	?	?	?
Cache kernel [21]	Low	?	15k	?
Dual VAX 11/780 [32]	High	Low	?	?
Application kernel [57]	High	Low	3,600	5 weeks
Piglet [76]	As UP	Dep. on UP	?	?
Cellular Disco [34]	As Guest	As Guest	50k	?
VMWare ESX [90]	As Guest	As Guest	?	?
L4Linux [102]	As Guest	As Guest	?	?
Adeos [107]	As Guest	As Guest	?	?
Xen [12]	As Guest	As Guest	75k+38k	?
K42 [6]	Low	High	50k	?

2.4.2 Lock-free Implementations

To our knowledge, there exists only two operating system kernels which rely solely on lock-free algorithms; Synthesis [68] and the Cache Kernel [21]. Synthesis uses a traditional monolithic structure but restricts kernel data structures to a few simple lock-free implementations of e.g., queues and lists.

The Cache Kernel [21] (Figure 2.1g), was also implemented lock-free. The Cache Kernel provides basic kernel support for address spaces, threads, and application kernels. Instead of providing full implementations of these concepts, the Cache Kernel caches a set of active objects which are installed by the application kernels. For example, the currently running

threads are present as thread objects holding the basic register state, while an application kernel holds the complete state. The Cache Kernel design caters for a very small kernel which can be easily verified and implemented in a lock-free manner. Note, however, that the application kernels still need to be parallelized to fully benefit from multiprocessor operation. Both Synthesis and the Cache Kernel were implemented for the Motorola 68k CISC architecture, which has architectural support for DCAS. Implementations for other architectures which lack DCAS support might be more difficult.

2.4.3 Asymmetric Implementations

The most common asymmetric systems have been master-slave systems [32], which employ one master processor to run kernel code while the other ("slave") processors only execute user space applications. The changes to the original OS in a master-slave port are mainly the introduction of separate queues for master and slave jobs, as shown in Figure 2.1d. Like giant locks, the performance of master-slave systems is limited by allowing only one processor in the kernel.

The Application kernel approach [57] (Figure 2.1e) allows keeping the original uniprocessor kernel as-is. The approach runs the original unmodified kernel on one processor, while user-level applications run on a small custom kernel on the other processors. All processes are divided in two parts, application threads and one bootstrap thread. The application threads run the original application, while the bootstrap thread forwards system calls, page faults etc., to the uniprocessor kernel.

Piglet [76] (Figure 2.1f) dedicates the processors to specific operating system functionality. Piglet allocates processors to run a Lightweight Device Kernel (LDK), which normally handles access to hardware devices but can perform other tasks. The LDK is not interrupt-driven, but instead polls devices and message buffers for incoming work. A prototype of Piglet has been implemented to run beside Linux 2.0.30, where the network subsystem (including device handling) has been offloaded to the LDK, and

the Linux kernel and user-space processes communicate through lock-free message buffers with the LDK.

2.4.4 Virtualization

There are a number of virtualization systems. VMWare ESX server [90] is a fully virtualized system which uses execution-time dynamic binary translation to handle the deficiencies of the IA-32 platform. Cellular disco [34] is a paravirtualized system created to use large NUMA machines efficiently. The underlying Hypervisor is divided into isolated cells, each handling a subset of the hardware resources to provide fault containment.

Xen [12] uses a paravirtualized approach for the Intel IA-32 architecture currently capable of running uniprocessor Linux and NetBSD as guest OS:es. The paravirtualized approach allows higher performance on IA-32. For example, the real hardware MMU can be used instead of software lookup in a virtual MMU. The Xen hypervisor implementation consists of around 75,000 lines of code while the modifications to Linux 2.6.10, mostly being the addition of a virtual architecture for Xen, is around 38,000 lines of code. The Adeos Nanokernel [107] also works similar to Xen, requiring modifications to the guest kernel's (Linux) source code.

2.4.5 Reimplementation

As an example of a reimplementation, K42 [6] (Figure 2.1i) is ABI-compatible with Linux but implemented from scratch. K42 is a microkernel-based operating system with most of the operating system functionality executing in user-mode servers or replaceable libraries. K42 avoids global objects and instead distributes objects among processors and directs access to the processor-local objects. Also, K42 supports runtime replacement of object implementations to improve performance for various workloads.

2.5 Linux Case Study

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

We have studied how three parameters have evolved from kernel versions 2.0 to 2.6. First, we examined the locking characteristics. Second, we examined the source code changes for multiprocessor support, and third, we measured performance in a kernel-bound benchmark.

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

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

2.5.1 Evolution of Locking in Linux

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

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

The 2.4 version of the kernel further relaxes the locking scheme. For example, the BKL is no longer held for virtual file system reads and writes. Like earlier versions, 2.4 employs a single shared runqueue from which all processors take jobs.

Many improvements of the multiprocessor support were added in the 2.6 release. 2.6 introduced seqlocks, read-copy update mutual exclusion [69], processor-local runqueues, and kernel preemption. Kernel preemption allows processes to be preempted within the kernel, which reduces latency. A seqlock is a variant of rwlocks that prioritizes writers over readers. Read-copy update, finally, was carried over from K42 and is used to defer updates to a structure until a safe state when all active references to that structure are removed, which allows for lock-free access. The safe state is when the process does a voluntary context switch or when the idle loop is run, after which the updates can proceed.

2.5.2 Locking and Source Code Changes

Table 2.2 shows the how the lock usage has evolved throughout the Linux development. The table shows the number of places in the kernel where locks are acquired and released. Semaphores (sema in the table) are often used to synchronize with user-space, e.g., in the system call handling, and thus have the same use on uniprocessors.

As the giant lock in 2.0 protects the entire kernel, there are only 17 places with BKL operations (on system calls, interrupts, and in kernel daemons). The coarse-grained approach in 2.2 is significantly more complex. Although 2.2 introduced a number of separate spinlocks, around 30% (over

Table 2.2: Number of locks in the Linux kernel.

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

250 places) of the lock operations still handle the giant lock. The use of the giant lock has been significantly reduced in 2.4, with around 13% of the lock operations handling the giant lock. This trend continues into 2.6, where less than 5% of the lock operations handled the giant lock. The 2.6 seqlocks and read-copy update mutual exclusion are still only used in a few places in the kernel.

Table 2.3 shows the results from the C preprocessor study. From the table, we can see that most files do not contain explicit changes for the multiprocessor support. In terms of modified, added, or removed lines, multiprocessor support for the 2.0 kernel is significantly less intrusive than the newer kernels, with only 541 source lines (1.19% of the uniprocessor source code) modified. In 2.2 and 2.4, around 2.2% of the lines differ between the uniprocessor and multiprocessor kernels, while the implementation is closer again in 2.6 with 1.7% of the lines changed.

2.5.3 Performance Evaluation

We also did a performance evaluation to compare the different Linux kernel versions in the Postmark benchmark [49]. The Postmark benchmark models the file system behavior of Internet servers for electronic mail and web-based commerce, focusing on small-file performance. This kernel-bound benchmark requires a highly parallelized kernel to exhibit performance improvements (especially for the file and block I/O subsystems). We ran Postmark with 10,000 transactions, 1,000 simultaneous files and a file size of 100 bytes.

Table 2.3: Lines of code with and without SMP support in Linux.

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

We compiled the 2.0 and 2.2 kernels with GCC 2.7.2 whereas 2.4 and 2.6 were compiled with GCC 3.3.5. All kernels were compiled with SMP-support enabled, which is a slight disadvantage on uniprocessors. The system is a minimal Debian GNU/Linux system which uses the ext2 filesystem. We ran 8 Postmark processes in parallel and measured the time used for all of them to complete. The benchmark was executed in the Simics full-system simulator [65], which was configured to simulate between 1 and 8 processors. Simics simulates a complete computer system including disks, network, and CPUs including the memory hierarchy modeled after the Pentium 4.

Figure 2.2 presents the scalability results for the Postmark benchmark normalized to uniprocessor performance in Linux 2.0. First, we can see that the absolute uniprocessor performance has increased, with 2.4 having the best performance. Linux 2.0 and 2.2 does not scale at all with this benchmark, while 2.4 shows some improvement over uniprocessor mode. On the other hand, 2.6 scales well all the way to 8 processors. Since the 2.0 kernels use the giant locking approach, it is not surprising that it does not scale in this kernel-bound benchmark. Since much of the file subsystem in 2.2 still uses the giant lock and no scalability improvement is shown. The file subsystem revision in 2.4 gives it a slight scalability advantage, although it does not scale beyond 3 processors. It is not until the 2.6 kernel that Linux manages to scale well for the Postmark benchmark. Linux 2.6 has a very good scalability, practically linear until 7 processors.

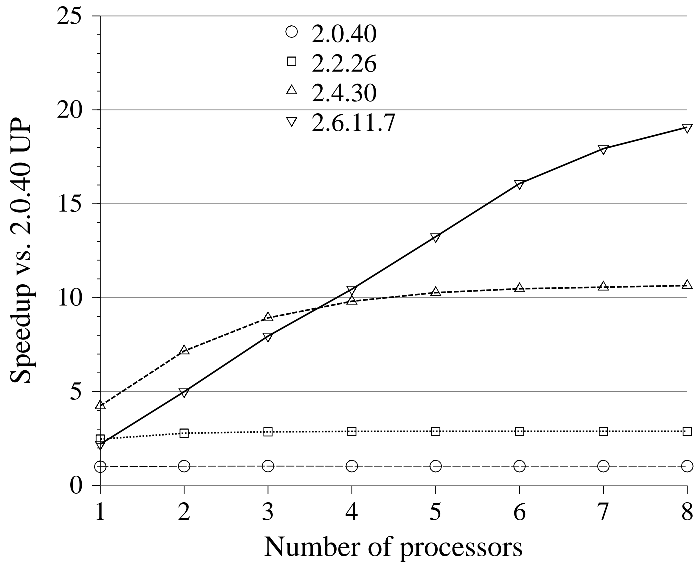


Figure 2.2: Postmark benchmark running on different versions of Linux.

2.6 Discussion and Conclusions

Figure 2.3 shows an approximation of the trade-off between scalability and effort of the categorizes presented. It should be noted that when porting an uniprocessor kernel to a multiprocessor, it is not always possible to freely select the porting approach. For example, employing the Xen hypervisor is only possible if the uniprocessor kernel is written for one of the architectures which Xen supports (currently IA-32).

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

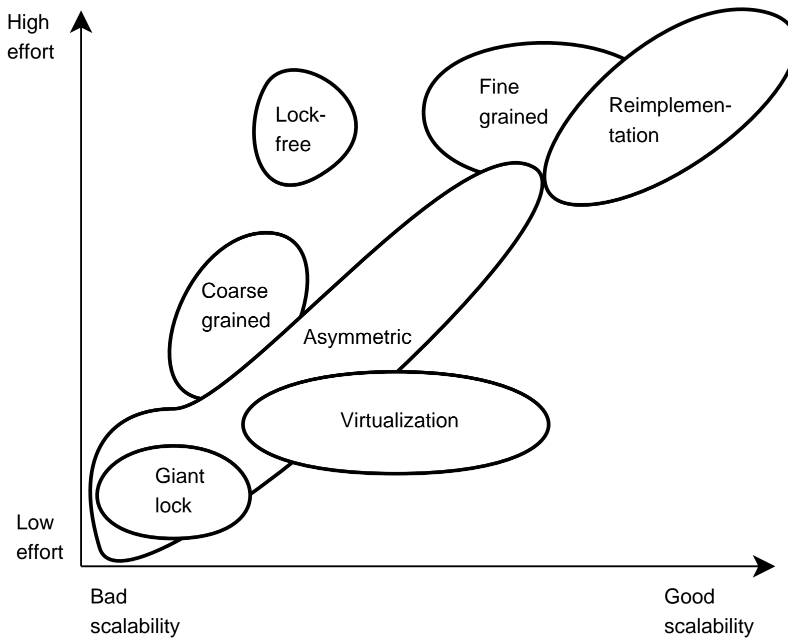


Figure 2.3: The available design space for multiprocessor implementations.

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

Since asymmetric systems are very diverse, both scalability and effort will vary depending on the approach. In one extreme, the application kernel provides a generic porting method with low effort at the cost of bad scalability for kernel-bound applications. Master-slave systems require more modifications to the original kernel, but have slightly better performance than the application kernel. More complex asymmetric systems, such as Piglet, can have good scalability on I/O-intensive workloads.

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

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

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

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

Chapter 3

Paper II

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

Simon Kågström, Håkan Grahn, Lars Lundberg

Submitted for journal publication, June 2005

3.1 Introduction

A current trend in the computer industry is the transition from uniprocessors to various kinds of multiprocessors, also for desktop and embedded systems. Apart from traditional SMP systems, many manufacturers are now presenting chip multiprocessors or simultaneous multithreaded CPUs [46, 67, 98] which allow more efficient use of chip area. The trend towards multiprocessors requires support from operating systems and applications to take advantage of the hardware.

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

We are working on a project together with a producer of large industrial systems in providing multiprocessor support for an operating system kernel. The operating system is a special-purpose industrial system primarily used in telecommunication systems. It currently runs on clusters of uniprocessor Intel IA-32 computers, and provides high availability and fault tolerance as well as (soft) real-time response time and high throughput performance. The system can run on one of two operating system kernels, either the Linux kernel or an in-house kernel, which is an object-oriented operating system kernel implemented in C++. The in-house kernel offers higher performance while Linux provides compatibility with third-party libraries and tools. With multiprocessor hardware becoming cheaper and more cost-effective, a port to multiprocessor hardware is becoming increasingly interesting to harvest the performance benefits of the in-house kernel.

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

The rest of the paper is structured as follows. Section 3.2 describes the structure and the programming model for the operating system. Section 3.3 thereafter describes the design decisions made for the added multiprocessor support. Section 3.4 outlines the method we used for evaluating our implementation, and Section 3.5 describes the evaluation results. We thereafter discuss some experiences we made during the implementation

in Section 3.6 and describe related and future work in Section 3.7. Finally, we conclude in Section 3.8.

3.2 The Operating System

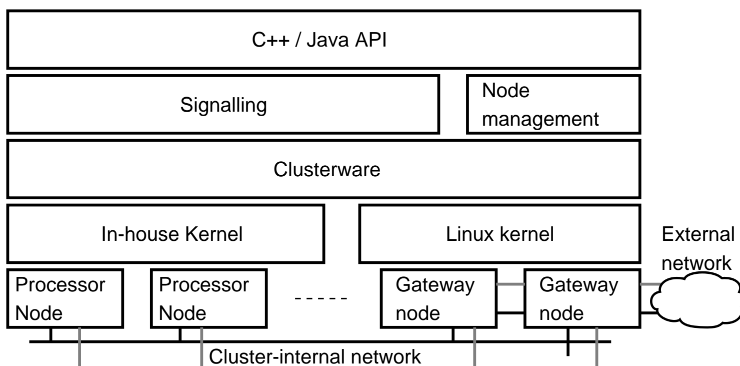


Figure 3.1: The architecture of the in-house operating system.

Figure 3.1 shows the architecture of the operating system. The system exports a C++ or Java API to application programmers for the clusterware. The clusterware runs on top of either the in-house kernel or Linux and provides access to a distributed RAM-resident database, cluster management that provides fail-safe operation and an object broker (CORBA) that provides interoperability with other systems.

A cluster consists of processing nodes and gateway machines. The processing nodes handle the workload and usually run the in-house kernel. Gateway machines run Linux and act as front-ends to the cluster, forwarding traffic to and from the cluster. The gateway machines further provide logging support for the cluster nodes and do regular backups to hard disk of the database. The cluster is connected by redundant Ethernet connections internally, while the connections to the outside world can be either SS7 [45] or Ethernet. Booting a node is performed completely over the network by PXE [43] and TFTP [92] requests handled by the gateway machines.

3.2.1 The Programming Model

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

In the operating system, two types of processes, *static* and *dynamic*, are defined. Static processes are restarted on failure and can either be unique or replicated in the system. For unique static processes, there is only one process of that type in the whole system, whereas for replicated processes, there is one process per node in the system. If the node where a unique process resides crashes, the process will be restarted on another node in the system. Replicated static processes allow other processes to communicate with the static process on the local node, which saves communication costs.

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

3.2.2 The Distributed Main-Memory Database

The operating system employs an object-oriented distributed RAM-resident database which provides high performance and fail-safe operation. The database stores persistent objects which contain data and have methods just like other objects. The objects can be accessed transparently across

nodes, but local objects are faster to access than remote ones (which is the reason to tie processes to database objects).

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

3.2.3 The Process and Memory Model

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

To allow for the asynchronous programming model and short-lived processes, the in-house kernel supplies very fast creation and termination of processes. There are several mechanisms behind the fast process handling. First, each code package (object code) is located at a unique virtual address range in the address space. All code packages also reside in memory at all times, i.e., similar to single-address space operating systems [19, 38]. This allows fast setup of new containers since no new memory mappings are needed for object code. The shared mappings for code further means that there will never be any page faults on application code, and also that RPC can be implemented efficiently.

The paging system uses a two-level paging structure on the IA-32. The first level on the IA-32 is called a *page directory* and is an array of 1024 *page directory entries*, each pointing to a *page table* mapping 4MB of the address space. Each *page table* in turn contains *page table entries* which describe the mapping of 4KB virtual memory pages to physical memory

pages. During kernel initialization, a global page directory containing application code and kernel code and kernel data is created, and this page directory then serves as the basis for subsequent page directories since most of the address space is identical between containers. The address space of the in-house kernel is shown in Figure 3.2.

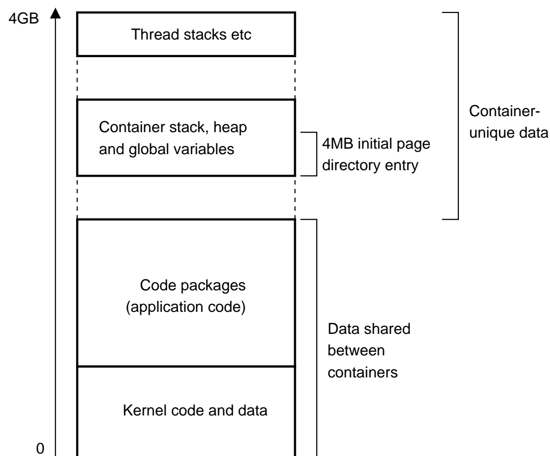


Figure 3.2: The in-house kernel address space on Intel IA-32 (simplified).

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

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

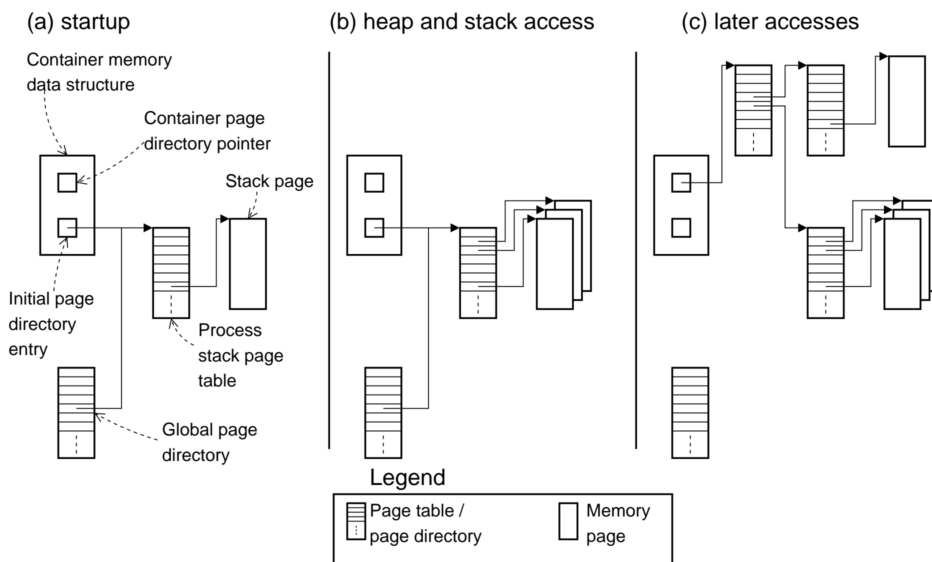


Figure 3.3: Handling of container address spaces in the in-house kernel

Figure 3.3 shows the container address space handling in the operating system. In Figure 3.3a, the situation right after process startup is shown. The container first uses the global page directory, with two pages allocated: one for the stack page table and one for the process stack. This situation gradually evolves into Figure 3.3b, where the process has allocated more pages for the stack, the heap or global variables, still within the 4MB area covered by the stack page table. When the process accesses data outside the stack page table, the global page directory can no longer be used and a new page directory is allocated and copied from the global as shown in Figure 3.3c.

3.3 Design of the Multiprocessor Support

In this section, we discuss the design of multiprocessor support for the in-house kernel. We describe the locking scheme we adopted, the imple-

mentation of CPU-local data, and optimizations made possible by the special properties of the in-house kernel.

3.3.1 Kernel Locking and Scheduling

For the first multiprocessor implementation, we employ a simple locking scheme where the entire kernel is protected by a single, “giant” lock (see Chapter 10 in [91]). The giant lock is acquired when the kernel is entered and released again on kernel exit. The advantage of the giant locking mechanism is that the implementation is kept close to the uniprocessor version. Using the giant lock, the uniprocessor semantics of the kernel can be kept, since two CPUs will never execute concurrently in the kernel. For the initial version, we deemed this important for correctness reasons and to get a working version early. However, the giant lock has shortcomings in performance since it locks larger areas than potentially needed. This is especially important for kernel-bound processes and multiprocessors with many CPUs. Later on, we will therefore relax the locking scheme to allow concurrent access to parts of the kernel.

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

3.3.2 CPU-local Data

Some structures in the kernel need to be accessed privately by each CPU. For example, the currently running thread, the current address space, and the kernel stack must be local to each CPU. A straightforward method of solving this would be to convert the affected structures into vectors, and

index them with the CPU identifier. However, this would require extensive changes to the kernel code, replacing every access to the structure with an index-lookup. It would also require three more instructions (on IA-32) for every access, not counting extra register spills etc.

This led us to adapt another approach instead, where each CPU always runs in a private address space. With this approach, each CPU accesses the CPU-local data at the same virtual address without any modifications to the code, i.e., access of a CPU-local variable is done exactly as in the uniprocessor kernel. To achieve this, we reserve a 4KB virtual address range for CPU-local data and map this page to different physical pages for each CPU. The declarations of CPU-local variables and structures are modified to place the structure in a special ELF-section [101], which is page-aligned by the boot loader.

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

3.3.3 Multithreaded Processes

The CPU-local page presents a problem for multithreaded containers (address spaces). Normally, these would run in the same address space, which is no problem on a uniprocessor system. In a multiprocessor system, however, using a single address space for all CPUs would cause the CPU-local virtual page to map to the same physical page for all CPUs, i.e., the CPU-local variables would be the same for all CPUs. To solve this problem, a multithreaded container needs a separate page directory for every CPU which executes threads in the container. However, we do not want to

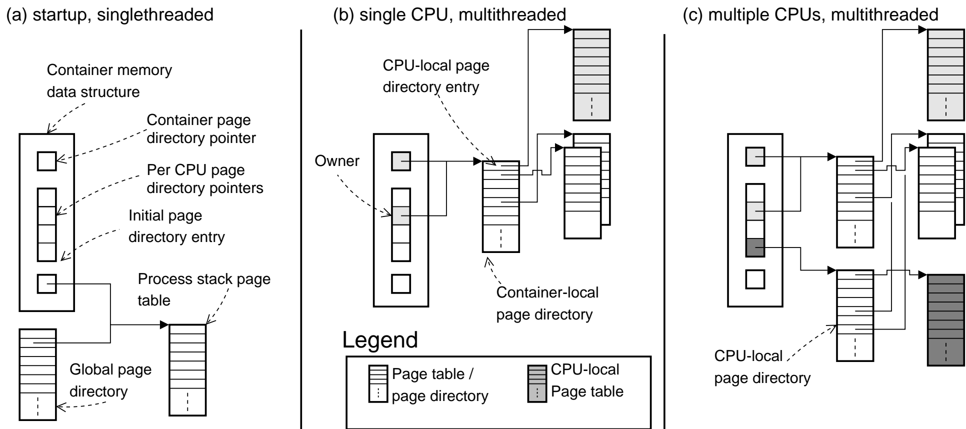


Figure 3.4: Handling of container address spaces in the in-house kernel for multiprocessor computers

compromise the low memory requirements for containers by preallocating a page for every CPU.

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

Figure 3.4 shows the handling of multithreaded containers on multiprocessors in the in-house kernel. The figure shows the container memory data structure, which has a container page directory pointer and an initial page directory entry as before (see Figure 3.3 and Section 3.2.3), but has also been extended with an array of per-CPU page directory pointers.

When the process starts up it will have only one thread and the situation is then as in Figure 3.4a. The process initially starts without a private address space and instead uses the global address space (which is CPU-local). The global page directory is modified with a page table

for the process stack, global variables and part of the heap. As long as the process is singlethreaded and uses moderate amounts of heap or stack space, this will continue to be the case.

When the process becomes multithreaded the first time, as shown in Figure 3.4b, a new *container page directory* is allocated and copied from the global page directory[1]. The current CPU will then be set as the owner of the container page directory. The CPU-local entry of the page directory is thereafter setup to point to the CPU-local page table of the CPU that owns the container page directory. Apart from setting the owner, this step works exactly as in the uniprocessor version. Since the thread stacks reside outside the 4MB process stack area, multithreaded processes will soon need a private address space, so there is no additional penalty in setting up the address space immediately when the process becomes multithreaded.

As long as only one CPU executes the threads in the process, there will be only one page directory used. However, as soon as another CPU schedules a thread in the process, a single page directory is no longer safe. Therefore, the container page directory is copied to a new CPU-local page directory which is setup to map the CPU-local page table. This is shown in Figure 3.4c. Note that apart from the CPU-local page table, all other page tables are identical between the CPUs. When scheduling the thread, the CPU-local page directory will be used.

One complication with this scheme is page fault handling. If two or more CPUs run in a container, a page fault will be generated for the CPU-local page directory. We therefore modified the page fault handler to always update the container page directory beside the CPU-local page directory. However, there can still be inconsistencies between page directories if the owner of the container page directory causes a page fault, which would only update the container page directory. A later access on the same page from another CPU will then cause a spurious page fault. We handle this situation lazily by checking if the page was already mapped in the container page directory, in which case we just copy the entry to the faulting page directory. Note that this situation is fairly uncommon

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

since it only affects faults on unmapped page directories, i.e., 4MB areas. Faults on 4KB pages will be handled transparently of our modifications since the page tables are shared by all CPUs.

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

3.4 Evaluation Framework

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

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

We also benchmarked the locking scheme to see the proportion of time spent in holding the giant lock, spinning for the lock, and executing with-

out the lock (i.e., executing user-level code). The locking scheme was benchmarked by instrumenting the acquire lock and release lock procedures with a reading of the CPU cycle counter. The lock time measurement operates for one CPU at a time, in order to avoid inconsistent cycle counts between the CPUs and to lessen the perturbation from the instrumentation on the benchmark. The locking scheme is measured from the start of the benchmark application until it finishes.

3.5 Evaluation Results

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

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

Table 3.1: Proportion of time spent executing user and kernel code.

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

Table 3.1 shows the lock contention when the benchmark application run completely in user-mode, both the uniprocessor and the multiproces-

sor. For the uniprocessor, acquiring the lock always succeeds immediately. From the table, we can see that the uniprocessor spends around 36% of the time in the kernel. On the multiprocessor, all times are shared between two CPUs, and we see that 20%-23% of the time is spent spinning for the giant lock. Since the in-kernel time is completely serialized by the giant lock, the theoretically maximum speedup we can achieve on a dual processor system is $\frac{36+64}{36+\frac{64}{2}} \approx 1.47$ according to Amdahl's law.

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

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

3.6 Implementation Experiences

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

First, the development of a multiprocessor kernel is generally harder then a uniprocessor kernel because of inherent mutual exclusion issues. We therefore wanted to perform the development in the Simics full-system

simulator [65], and a related project investigated running the operating system on Simics. It turned out, however, that it was not possible at that time to boot the system on Simics because of lacking hardware support in Simics. Second, we performed most of the implementation off-site, which made it harder to get assistance from the core developers. Coupled to the fact that the system is highly specialized and complex to build and setup, this led us to spend significant amount of time on configuration issues and build problems. Finally, the code base of the operating system is large and complex. The system consists of over 2.5 million lines totally, of which around 160,000 were relevant for our purposes. The complexity and volume of the code meant that we had to spend a lot of time to grasp the functionality of the code.

In the end, we wrote around 2,300 lines of code in new files and modified 1,600 existing lines for the implementation. The new code implement processor startup and support for the locking scheme whereas the modified lines implement CPU-local data, acquiring and releasing the giant lock etc. The changes to the original code is limited to around 1% of the total relevant code base, which shows that it is possible to implement working multiprocessor support with a relatively modest engineering effort. We chose the simple giant lock to get a working version fast and the focus is now on continuous improvements which we discuss in Section 3.7.

3.7 Related and Future Work

The operating system studied in this paper has, as mentioned before, a number of properties that are different from other cluster operating systems. It provides a general platform with high availability and high performance for distributed applications and an event-oriented programming environment based on fast process handling. Most other platforms and programming environments are mainly targeted at high performance and/or parallel and distributed programming, e.g., MPI [70] or OpenMP [80]. These systems run on networked computer nodes running a standard operating system, and are not considered as cluster operating systems.

There exists some distributed operating systems running on clusters of Intel hardware. One such example is Plurix [33], which has several similarities with the operating system. Plurix provides a distributed shared memory where communication is done through shared objects. The consistency model in Plurix is based on restartable transactions coupled with an optimistic synchronization scheme. The distributed main memory database in the operating system serves the same purpose. However, to the best of our knowledge, Plurix only runs on uniprocessor nodes and not on multiprocessors in a cluster. Plurix is also Java-based whereas the operating system presented in this paper supports both C++ and Java development.

Many traditional multiprocessor operating systems have evolved from monolithic uniprocessor kernels, e.g., Linux and BSD. Such monolithic kernels contain large parts of the actual operating system which make multiprocessor adaptation a complex task. Early multiprocessor operating systems often used coarse-grained locking, for example using a giant lock [91]. The main advantage with the coarse-grained method is that most data structures of the kernel can remain unprotected, and this simplifies the multiprocessor implementation. For example, Linux and FreeBSD both initially implemented giant locks [13, 60].

For systems which have much in-kernel time, the time spent waiting for the kernel lock can be substantial, and in many cases actually unnecessary since the processors might use different paths through the kernel. Most evolving multiprocessor kernels therefore moves toward finer-grained locks. The FreeBSD multiprocessor implementation has for example shifted toward a fine-grained method [60] and mature UNIX systems such as AIX and Solaris implement multiprocessor support with fine-grained locking [22, 50], as do current versions of Linux [63].

Like systems which use coarse-grained locking, master-slave systems (refer to Chapter 9 in [91]) allow only one processor in the kernel at a time. The difference is that in master-slave systems, one processor is dedicated to handling kernel operations (the “master” processor) whereas the other processors (“slave” processors) run user-level applications and only access the kernel indirectly through the master processor. Since all

kernel access is handled by one processor, this method limits throughput for kernel-bound applications.

In [57], an alternative porting approach focusing on implementation complexity is presented. The authors describe the *application kernel approach*, whereby the original uniprocessor kernel is kept as-is and the multiprocessor support is added as a loadable module to the uniprocessor kernel. This allows the uniprocessor kernel to remain essentially unchanged, avoiding the complexity of in-kernel modifications. The approach is similar to master-slave systems performance-wise since all kernel operations are performed by one processor in the system. Neither the master-slave approach nor the application kernel approach provide any additional performance benefit over our giant lock, and incrementally improving the giant locking with finer-grained strategies is easier.

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

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

dividing the common ready queue into one queue per processor, which is done in for example Linux 2.6 [63].

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

3.8 Conclusions

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

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

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

Chapter 4

Paper III

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

Simon Kågström, Håkan Grahn, Lars Lundberg

Submitted for journal publication, June 2005

4.1 Introduction

For performance reasons, uniprocessor computers are now being replaced with small multiprocessors. Moreover, modern processor chips from major processor manufacturers often contain more than one CPU core, either logically through Symmetric MultiThreading [26] or physically as a Chip MultiProcessor [37]. For instance, current Intel Pentium 4 and Xeon processors contain two logical processors [67] and several other manufacturers are in the process of introducing on-chip multiprocessors [46, 93].

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 being 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 [53]. 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. [7] 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 [109].

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 [57].

The rest of the paper is structured as follows. We begin with discussing related work in Section 4.2. In Section 4.3 we describe the ideas behind our approach and Section 4.4 then discusses our implementation for the Linux kernel. We describe our evaluation framework in Section 4.5, and then evaluate the implementation complexity and performance of the application kernel in Section 4.6. Finally, we conclude and discuss future extensions to the approach in Section 4.7.

4.2 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.

4.2.1 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 [8]. These systems employ a locking scheme where only one processor runs in the kernel (or in a kernel subsystem) at a time [91]. 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 [59] and mature UNIX systems such as AIX and Solaris implement multiprocessor support with fine-grained locking [22, 50], as do current versions of Linux [63].

4.2.2 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 [40], a modified Linux kernel which runs on top of the L4 microkernel [61]. The Mach microkernel has been used as the base for many operating systems, for example DEC OSF/1 [23] and MkLinux [24]. Further, QNX [84] 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 [17, 89]. 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 [30, 85] 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.

4.2.3 Asymmetric Operating Systems

Like systems which use coarse-grained locking, master-slave systems (refer to Chapter 9 in [91]) 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 [32] 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 [76]. Piglet [76] 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 [86]. 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.

4.2.4 Cluster-based Approaches

Several approaches based on virtualized clusters have also been presented. One example is the Adeos Nanokernel [107] 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 [12], which virtualizes Linux or NetBSD systems.

Another cluster-based method is Cellular Disco [34], 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 [11] 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).

4.3 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.

4.3.1 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.

4.3.2 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 4.1 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 *boot-*

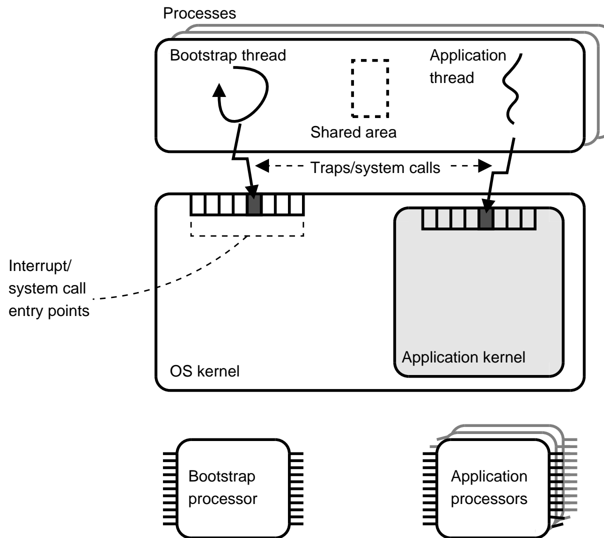


Figure 4.1: Overview of the application kernel approach.

strap 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 4.3.4.

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 4.4.1 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 [36, 21] employs a scheme similar to ours on redirecting system calls and page faults, but requires a complete reimplementation of the original kernel to adapt it to the cache kernel. We can also compare it to the MOSIX system [11] which also redirects system calls, although MOSIX is used in a cluster context and has different goals then the application kernel.

4.3.3 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 [74] and PowerPC has a decrementer register [41] 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.

4.3.4 Application Kernel Interaction

Figure 4.2 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.

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.

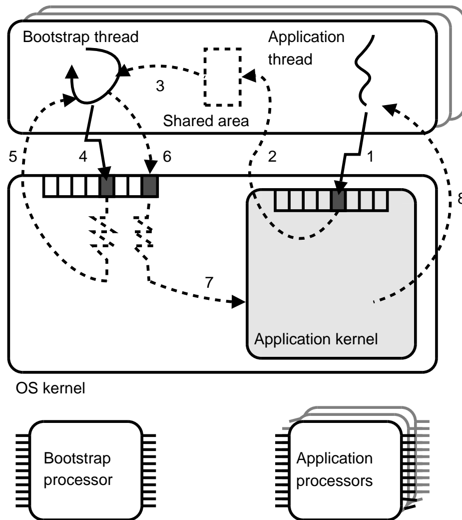


Figure 4.2: System call/trap handling in the application kernel approach

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 then other calls, and are described in detail in Section 4.4.2. 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.

4.3.5 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*.

4.4 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 4.3. 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.

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 4.4.3). All interactions with the application kernel, apart from `open` and `close`, are done using `ioctl` calls, through which the exported interface is available.

Figure 4.4 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.

```
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 4.3: Application kernel shared area layout

4.4.1 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 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 multithreaded processes on page table updates, when the TLB contents for different processors running in the same address space will be inconsis-

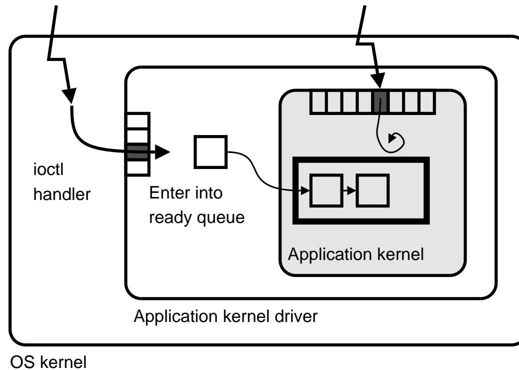


Figure 4.4: Application kernel device driver structure

tent[1]. 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.

4.4.2 `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

[1] On 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.

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 4.5. 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.

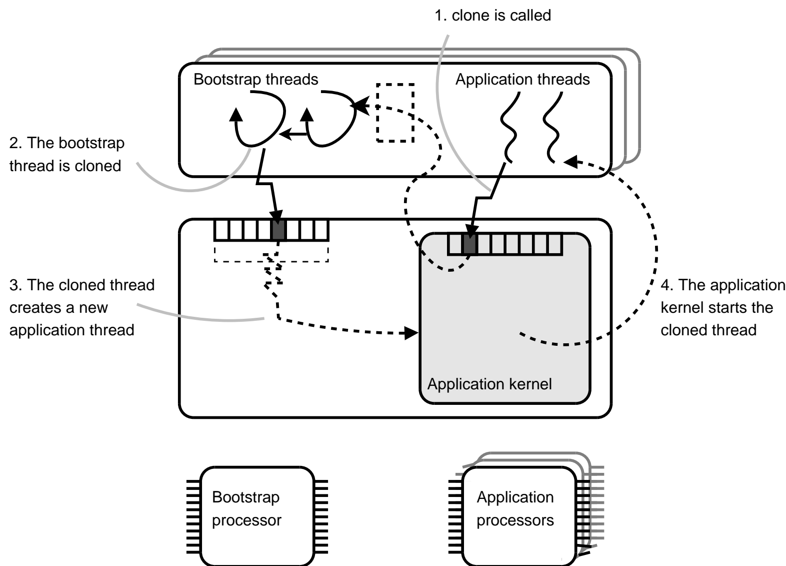


Figure 4.5: Handling of the `clone` system call

4.4.3 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.

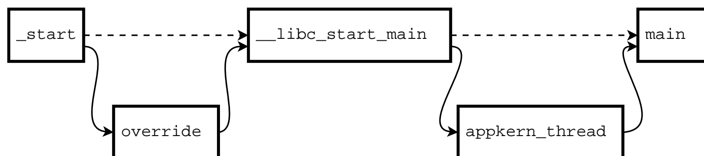


Figure 4.6: Application startup. The dashed lines shows the original execution while the solid lines show the overridden execution path.

The overriding process is illustrated in Figure 4.6. 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.

4.5 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 [106] benchmarks and the SPEC CPU2000 [94] 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 [27] which gives the number of independent code paths through a function. The code lines were counted with the sloccount tool [104] by David A. Wheeler and the cyclomatic complexity was measured by the pmccabe tool by Paul Bame [10].

4.5.1 Evaluation Environment

We performed our performance evaluation using the Simics full system simulator [65] 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 [28], 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.

4.5.2 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 [51] 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

Table 4.1: The benchmarks used in the performance evaluation

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

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 4.1.

4.6 Experimental Results

In this Section, we describe the results obtained from our measurements. Table 4.3 and 4.4 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 4.7. The results from the `getpid` evaluation is shown in Table 4.2.

4.6.1 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 4.2. 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 4.2). 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 4.2: `getpid` latency in Linux and the application kernel

	Linux	Application Kernel
PPro 200MHz	970	5700
Simics	74	860

Table 4.3: Speedup for the single-process benchmarks.

	Speedup vs. uniprocessor Linux					
	Find		176.gcc		164.gzip	
Procs.	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

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 4.3. 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 4.4 and Figure 4.7). The results

Table 4.4: Speedup for the parallel and multiprogramming benchmarks.

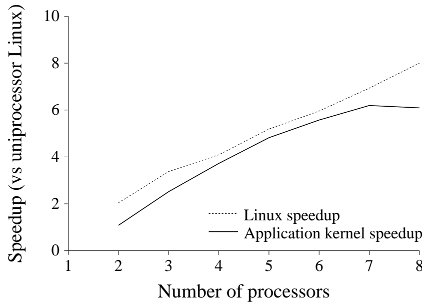
	Speedup vs uniprocessor Linux							
	RADIX		FFT		LU		Multiprogramming	
Procs.	Linux	Appkern	Linux	Appkern	Linux	Appkern	Linux	Appkern
2	2.0433	1.0834	1.6916	1.0401	1.9217	1.2662	1.5049	0.9705
3	3.3758	2.5174	2.2930	1.8654	2.9430	2.0795	1.6627	1.1375
4	4.0885	3.7227	2.5090	2.3235	3.5053	2.9941	1.6850	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	

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.

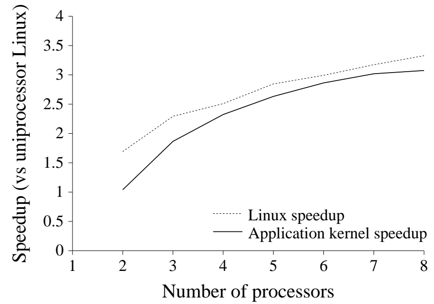
The multiprogramming benchmark, also shown in Table 4.4 and Figure 4.7, contains a mix of applications which have different behavior in terms of user/kernel execution. For this benchmark, we see that running all applications on the application kernel places a high strain on the bootstrap kernel, which 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.

4.6.2 Implementation Complexity and Size

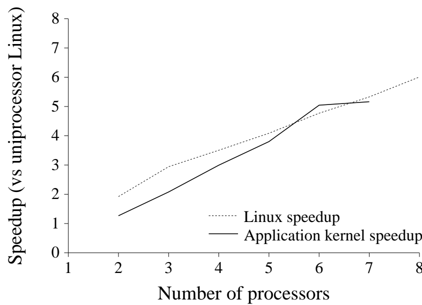
The application kernel was ported from the implementation presented in [57], and most of the internals of the kernel are completely unchanged. Apart from some restructuring and the loadable Linux kernel module, the only changes to the actual application kernel is some low-level handling of system calls (i.e., the used trap vector and parameter passing). One single developer spent seven weeks part-time implementing the application



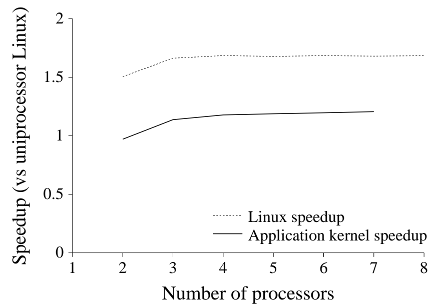
RADIX



FFT



LU



Multiprogramming

Figure 4.7: Speedup for the parallel and multiprogramming benchmarks vs. uniprocessor Linux.

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 4.5.

Table 4.5: 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 [48] is shown in Figure 4.8. 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.

4.7 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.

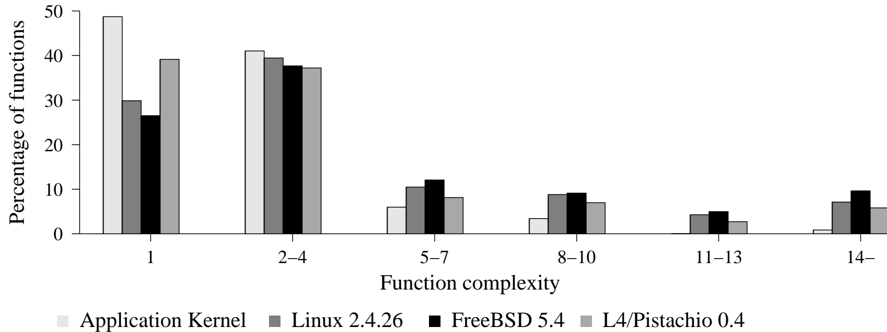


Figure 4.8: Histogram of McCabe cyclomatic complexity for the Application Kernel, Linux 2.4.26, FreeBSD 5.4 and the L4/Pistachio 0.4 microkernel.

Our approach has lower implementation complexity then 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.

Availability

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`.

Chapter 5

Paper IV

Automatic Low Overhead Program Instrumentation with the LOPI Framework

Simon Kågström, Håkan Grahn, Lars Lundberg

Published at the 9th Workshop on Interaction between Compilers and Computer Architectures, San Francisco, USA, February 2005

5.1 Introduction

Program instrumentation is a technique used in many and diverse areas. Instrumentation is often added to programs in order to investigate performance aspects of the applications [72, 88] as a complement to statistical profiling such as gprof [35], Intel VTune [105], or the Digital Continuous Profiling framework [5]. Instrumentation is also useful in many other areas not directly related to performance analysis, for instance call graph

tracing [96], path profiling [9], reversible debugging [20], code coverage analysis, and security [73].

Often, instrumentation is added manually by annotating the source code with instrumentation points. This task, however, is time-consuming, repetitive and error-prone, and it is both tied to the high-level language and access to source code. Over the years, there has therefore been a number of proposals to alleviate this situation. Today, there exists several libraries, e.g., ARM [79] and PAPI [62], which allows code-reuse for the instrumentation. There are also packages that provide graphical interfaces to select instrumentation-points and several tools for patching program binaries or relocatable object files [72, 58].

Another problem with program instrumentation is program behavior perturbations caused by the instrumentation [66, 75]. Regardless of how instrumentation is implemented, it always adds extra work for the program by affecting compiler optimizations (changed register allocation, reduced inlining possibilities etc.), altering the data reference patterns, and changing the execution flow. Taken together, these perturbations can cause the instrumented program to exhibit a substantially different behavior than the uninstrumented program. This problem is especially severe for performance instrumentation since the instrumented program should accurately reflect the uninstrumented program, and it is therefore important to measure and minimize the instrumentation overhead. The measurement itself can also be a problem, however. Although it is easy to measure the aggregate overhead of instrumenting a program, observing the detailed behavior of the instrumentation is harder since any performance measurement affects the program execution. Taken together, these problems lead us to we believe that it is important to explore optimizations for instrumentation, especially for frequently performed operations.

In this paper, we present the LOPI (LOw Perturbation Instrumentation) framework that provides a generic and easily used framework for instrumenting programs. In LOPI, we try to optimize for common instrumentation patterns in order to provide low perturbation on the program behavior. LOPI rewrites binary ELF-files for GNU/Linux on the IA-32 architecture in order to instrument an application. The current implemen-

tation instruments function entry and exit, but the approach is expandable to instrument most points in the code.

We provide measurements of the instrumentation perturbation using both real hardware and full-system simulations of seven SPEC CPU2000 benchmarks. We compare the LOPI framework to Dyninst[16] and regular source-based instrumentation. We find that source-based instrumentation usually has the lowest instrumentation overhead, on average executing 13% more instructions (5% inlined) for the studied applications, but with more tedious work for instrumenting the code. Comparing LOPI and Dyninst we find that LOPI has lower instruction overhead then Dyninst, on average 36% instruction overhead compared to 49% for Dyninst. Comparing the total execution times, we find that source-based instrumentation has 6% overhead, LOPI has 22% overhead, and Dyninst 28% overhead as compared to an uninstrumented application.

The rest of the paper is organized as follows. In Section 5.2 we provide an overview of program instrumentation, which is followed by an introduction of the LOPI framework in Section 5.3. In Section 5.4 we present the measurement methodology and in Section 5.5 we provide the measurement results. Finally, we discuss related work in Section 5.6 and conclude our findings in Section 5.7.

5.2 Background

5.2.1 Instrumentation approaches

Instrumentation packages can be grouped into three broad categories with different characteristics: *source-based instrumentation*, *binary rewriting*, and *memory image rewriting*. There are some special cases, for instance instrumentation at the assembly level, but these can normally be generalized into one of the above (assembly-level instrumentation is similar to binary rewriting except that it avoids some issues with relocatable code). Also, some completely different approaches exist. Valgrind [78], for instance, allows instrumentation of unmodified programs. Valgrind works

by running programs in a virtual machine, translating IA-32 binary code to a intermediate language, applying instrumentation, and then translated back to IA-32 code again. Valgrind allows instrumenting unmodified programs, but also imposes a high runtime overhead due to the code translation. Another approach is to run the application in a simulator, which gives no perturbation to the actual application, but has issues with accuracy and speed. Next, we will briefly describe the different approaches.

1. **Source-based instrumentation**: Source-based instrumentation works by inserting instrumentation calls as statements in the application source code. This allows the compiler to optimize the instrumented code, but it also inherently produces a different behavior compared to the non-instrumented code because of disturbed register allocation, inlining, etc. Further, this approach is dependent on the high-level implementation language as well as direct access to the source code.

   This category encompasses both libraries for instrumentation, i.e., where instrumentation is inserted manually into the source code [62], mixed solutions [29], and tools with source-to-source conversion from a graphical interface [95].

2. **Binary rewriting**: By patching the executable or the relocatable files, the high-level source code of the application can remain untouched. This prevents the compiler from optimizing the instrumentation code in the context of the application source code, but this should also give a closer correspondence to the uninstrumented application. This approach is also independent of the high-level language of the application and can in principle be used on applications for which the source code is unavailable.

   Many instrumentation packages work this way, for instance ATOM [4] and EEL [58] for UNIX-based systems, Etch [88] and PatchWrx [18] for Windows NT systems, and the LOPI framework presented here.

3. **Memory image rewriting** A final approach is to patch the application in-core, i.e., after the program has been loaded into memory. This approach, used by Dyninst [16, 72], allows instrumentation to be added to and removed from the program during runtime.

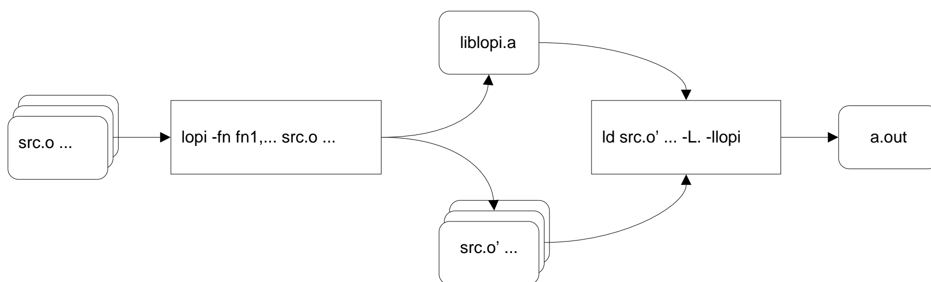


Figure 5.1: Overview of the instrumentation process. The functions and the files to instrument are given on the command line.

> The characteristics is similar to binary rewriting but memory image rewriting allows instrumentation to be dynamically removed when it is no longer needed, which can reduce unnecessary overhead.
>
> Memory image rewriting also adds some other interesting possibilities. Some programs, for instance operating system kernels cannot readily be restarted in order to have the instrumentation take effect. For these cases, memory image rewriting provides the only realistic alternative, and it has also been used for instrumentation of the Solaris [100] and Linux [82] kernels.

Each of these methods will cause perturbation to the application. Next we present an introduction to the various types of perturbation caused by instrumentation.

5.2.2 Instrumentation perturbation

Instrumentation perturbation is heavily dependent on the type of instrumentation applied. For performance instrumentation, the instrumentation might read a set of of hardware performance counters whereas call graph tracing requires significantly more complex operations [96]. Some parts are very common however. At the very basic end, instrumentation always causes more instructions to be executed, accesses more data in the

memory, and can also cause register spills. Further, there might be kernel invocations, device access or inter-process communication. The perturbation also varies over different phases of the program execution:

- **Initialization**: Most instrumentation packages have some sort of initialization phase. This can include, e.g., the initialization of hardware performance counters, creation of data structures, or memory image patching. This part can sometimes be very expensive, but is a one-time event.

- **Startup-phase**: During the first invocations of the instrumented code, the system will run with cold caches and need to bring the code and data into the caches.

- **Execution**: During the execution of the program, the instrumentation adds latency because more instructions are executed, increased cache pressure, and (potentially) extra kernel invocations.

- **End**: When the program ends, or the instrumentation is removed, the instrumentation package usually performs some cleanup operations (for instance freeing allocated memory, storing collected data on disk etc.). Like the initialization-phase, this is potentially expensive but normally has small effects on long-running programs.

For the execution phase, there are also some indirect effects on the execution that can arise from instrumentation. For instance, the addresses of data or executed instructions might change as a side-effect of instrumentation (this is especially likely with source instrumentation). The changed addresses can cause data or code to be aligned differently with respect to cache-lines, and also in some cases (albeit unusual) change actual program behavior [75]. In the LOPI framework, we have tried to minimize these effects by a number of optimizations, which are described in the next section.

5.3 The LOPI instrumentation framework

We have implemented an instrumentation package that tries to provide low and predictable overhead and still provide an easy interface to users. The framework uses the binary rewriting approach, although the ideas are applicable to memory rewriting (such as used by Dyninst) as well. Although we currently focus on function entry and exit, the approach is possible to combine with current methods for instrumentation at arbitrary points (still keeping the optimized entry/exit techniques). We have developed two types of performance instrumentations for LOPI, one utilizing the PAPI cross-platform front-end to performance counters [62] and one simple implementation measuring the processor cycle counter with the `rdtsc` instruction.

The process of instrumenting a program with the LOPI framework is shown in Figure 5.1. Using the LOPI framework adds one step in the compile process - running the LOPI executable after the relocatable files have been produced. The relocatable ELF-files are then linked with a library produced by LOPI at runtime, which contains stubs and the user-implemented instrumentation. Note that selecting the instrumentation points is done outside the LOPI framework in order to keep the framework general enough to support different kinds of instrumentation.

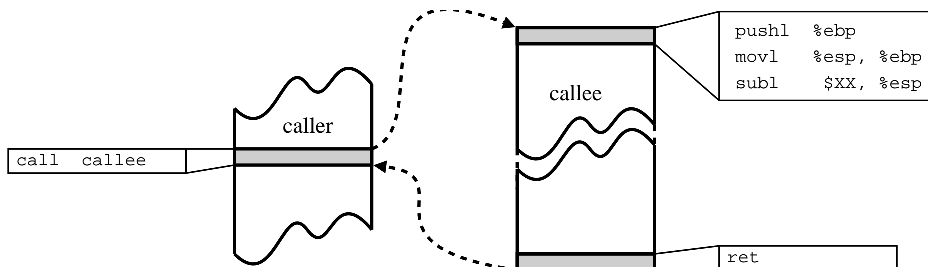


Figure 5.2: A non-instrumented function call.

Before going into details of the operation, we will first briefly describe the (GCC) calling convention for the IA-32 architecture. Figure 5.2 shows how *caller* calls the non-instrumented function *callee*. On IA-32, the `call`-

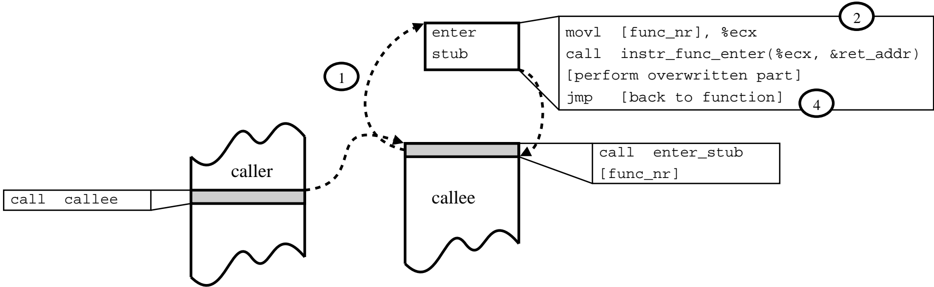


Figure 5.3: A function call instrumented with our approach.

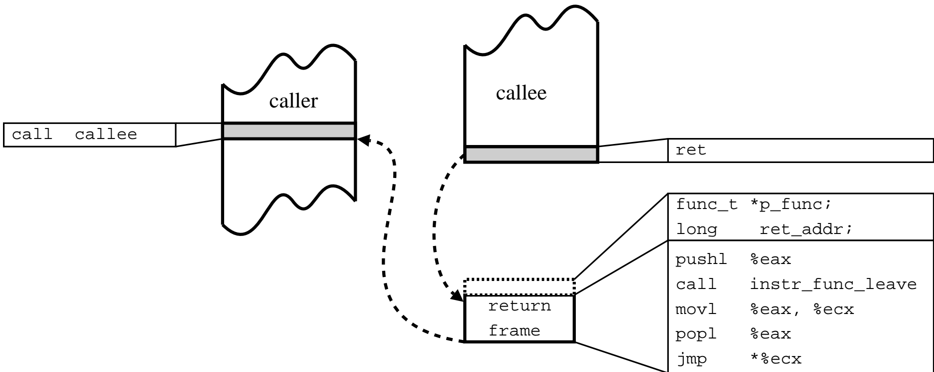


Figure 5.4: An instrumented function return.

instruction pushes the return address to the stack before switching to the function. On returning with `ret`, the instruction pointer is popped from the top of the stack. The IA-32 calling convention specifies that registers `%ebx`, `%edi`, `%esi`, and `%ebp` are callee-saved, whereas `%eax`, `%ecx` and `%edx` are caller-saved. Parameters are passed on the stack and the return value is passed in the `%eax` register. The function prologue shown initializes the function stack frame.

A function entry instrumented with the LOPI framework is shown, somewhat simplified, in Figure 5.3. When the program execution reaches

an instrumentation point, our library performs a four step operation. The sequence of events is shown in the figure and described below.

1. `enter_stub` is called (from *callee*) by the overwritten function prologue (which was replaced by the instrumentation). The `call`-instruction is immediately followed by an identifier for the function (`func_nr`). The function identifier defaults to a 8-bit value, but if more than 256 functions are instrumented this can be extended to a 16- or 32-bit value at instrumentation time (this has not yet been implemented, but the extension is simple to make).

2. `enter_stub` (shown in Figure 5.3) reads the function identifier (which is located at the return address, i.e., in the *callee*-prologue). Then, the enter stub calls `instr_func_enter`, which is common for all instrumented function entries.

```
struct ret_frame_t {
    func_t  *p_func
    long     ret_addr
    /* For icache/dcache conflict reduction */
    uint8_t  padding0[XX]
    uint8_t  program[16]
    uint8_t  padding1[XX]
    ...
}
ret_frame_t ret_frames[]

function instr_func_enter(func_nr, ret_addr) {
    /* Setup return frame */
    ret_frame = pop_ret_frame()
    ret_frame.func = funcs[func_nr]
    ret_frame.ret_addr = ret_addr
    ret_addr = ret_frame.program

    /* Perform the instrumentation */
    do_enter_func(func)
}
```

Figure 5.5: Pseudo code for the `instr_func_enter`-function.

3. The instr_func_enter-function, implemented in C (pseudo code in Figure 5.5), sets up a return frame to instrument the function return. inst_func_enter thereafter performs the actual instrumentation operation for function entries, which is implemented by the user of the instrumentation library and can be inlined. Access to the return frames is protected by a spinlock for multithreaded programs on SMPs.
4. After returning to the enter stub, the overwritten instructions of the function prologue are executed and the control returns to the function prologue (after the overwritten instructions).

There are some special cases for instrumenting function entry points, which suggest separate handling. First, we detect the common function prologue where the frame pointer (the %ebp register) is stored and a new stack frame is setup. This code sequence only varies with a constant, which gives the size of the new stack frame, and can therefore easily be represented by a common stub.

```
pushl %ebp       /* Save the old frame pointer */
movl  %esp, %ebp /* Set the start of the new frame */
subl   $XX, %esp /* Allocate stack space */
```

In the seven SPEC CPU2000 benchmarks we used (see Section 5.4), almost 80% of the function prologues had this pattern. This function prologue is represented with a special stub that stores the stack size XX. In the rare case that the function prologue is smaller than 6 bytes (the size of the call-instruction plus the function identifier) and the first basic block at the same time contains a branch target within the first 6 bytes, patching the function prologue is unsafe because the target instruction is overwritten. LOPI will detect and mark such areas as unavailable for instrumentation, although this functionality is only sketched in the prototype implementation.

Function returns are instrumented lazily with the return frames set up in instr_func_enter, i.e., without patching or adding source lines

to the program. The return frame is a data structure with the original return address (i.e., back to *caller* in this case), which also contains a machine code stub, copied to the structure at startup. The padding is needed since the return frame is accessed both as data and executed as code. Without the padding, the cache block (the stub is only 16 bytes) would ping-pong between the data and the instruction cache, degrading performance. The machine code stub acts as a trampoline for the function return instrumentation. The logic is as follows (refer to Figure 5.4):

1. The *callee* function returns with the `ret` instruction (i.e., exactly as without instrumentation). Since the return address was overwritten it will return to the return frame stub setup in `instr_func_enter`.

2. The return frame stub calls `instr_func_leave`. Since the position of the return frame (and thus the return stub) is unknown at compile-time, we need to do a register-relative call to `instr_func_leave` (not shown in the figure).

3. `instr_func_leave` performs the instrumentation on function exit (again specified by the user of the library), deallocates the return frame, and returns the original return address (i.e., to *caller* in this example). The pseudo code is shown in Figure 5.6.

For functions which modify the return address themselves, this optimization is unsafe, and a revert to a more traditional return instrumentation is needed. We reduce the perturbation of the instrumented application in a number of ways both during the program patching and during runtime:

1. **Inlined function identifiers**. The function identifier (shown in Figure 5.3) is placed directly in the instrumented code in order to avoid the need for calling separate stubs for every instrumentation point. The function identifier also allows us to lookup meta data for the instrumentation point by using it as a vector index instead of performing an expensive hash table lookup.

```
function instr_func_leave() {
    /* This code is contained in the ret_frame */
    ret_frame = [return address]-XX
    /* Perform the instrumentation */
    do_leave_func(ret_frame.func)

    push_ret_frame(ret_frame)
    /* Found in the ret_frame */
    return [original return address]

}
```

Figure 5.6: Pseudo code for the `instr_func_leave`-function.

2. **Code reuse**. A call-stub is shared for every instrumentation point with the same overwritten instructions. Also, the stubs are kept as short of possible with most of the logic in the generic enter and exit functions.

3. **Optimize for common cases**. We use a special stub for the common stack frame setup as explained in Section 5.3. This helps down the i-cache miss rate by reducing the number of instrumentation stubs.

4. **Register saving**. Our entry stubs does not store any registers for the function entries since we do not use any callee-saved registers in the stub. The return frame saves the `%eax` register since this is used for return values on IA-32.

5. **Data reuse**. The return frames are allocated in a stack-based scheme where the most recently used return frame is reused first.

The pollution of the instruction cache is limited by the number of function call stubs used in the instrumentation and the number of return frames used. The number of active return frames at a given point of time is equal to the current nesting depth of the instrumented functions, in most cases a fairly low number (the worst case occurs with deep recursion).

Table 5.1: Description of the SPEC CPU2000 benchmarks used in this study.

Benchmark	Description	Data set size
164.gzip	Compression	lgred.log
176.gcc	Compiler	smred.c-iterate.i
181.mcf	Combinatorial optimization	lgred.in
183.equake	Simulation of seismic wave propagation	lgred.in
197.parser	Grammar analysis	lgred.in
256.bzip2	Compression	lgred.graphic
300.twolf	CAD, Placement and global routing	lgred

Taken together, these optimizations significantly reduce the overhead of instrumentation. Further, since the call-stubs are aggressively reused, we expect the perturbation to be more predictable since less code is added to the program. The next section presents measurements comparing our approach to the Dyninst tool and basic source-based instrumentation.

5.4 Measurement methodology

For our measurements, we have used both real hardware and the Simics full-system simulator [65]. The machine we used is a Pentium III system running Linux, with a 1 GHz processor and 256 MB RAM. We use the hardware performance counters available on the Pentium III (through the PAPI [62] library) to capture the measures presented in Table 5.2, e.g., the number of instructions and cache misses.

As for our simulations, we simulate a complete Pentium III system with caches running a real operating system for performing the instrumentation measurements. The simulated system has 16 KB, 4-way set-associative, first-level data and instruction caches, and a unified 512KB, 8-way set-associative, second-level cache. Simics allows us to create a complete non-intrusive measurement of the application execution, both for instrumented

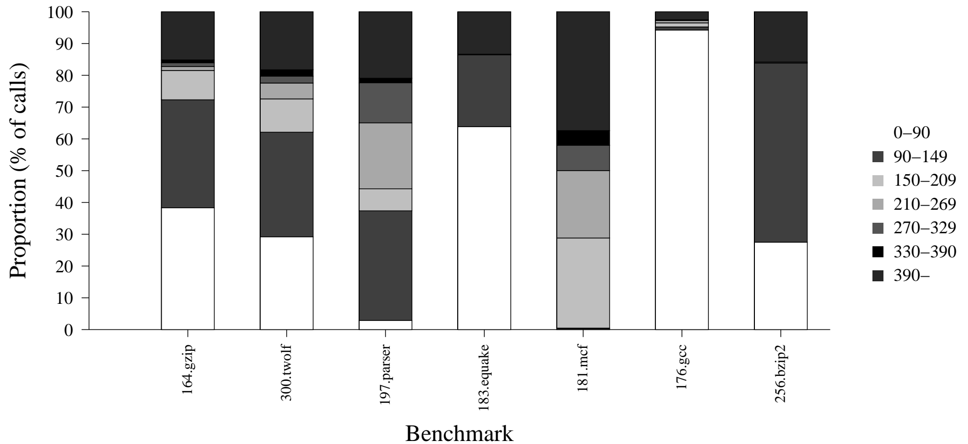


Figure 5.7: Cycles per function call on a subset of the SPEC CPU2000 benchmarks.

and non-instrumented applications. We can therefore isolate the impact of instrumentation from the application traces. We use Simics to provide detailed execution characteristics which were not possible to capture on real hardware, i.e., the figures in Figure 5.8.

We ran tests with seven applications from the SPEC CPU2000 benchmarks (compiled with GCC 2.95.4, optimization level -O3) on a minimal Linux 2.4.22-based system. A short description of the selected benchmarks is presented in Table 5.1. All measurements ran with the MinneSPEC [51] workloads in order to provide reasonable execution times in the simulated environment and each of the tests ran to completion. We chose to instrument the functions that make up 80% of the total execution time (as reported by gprof). Unfortunately, with Dyninst we were unable to instrument three of the applications when running on real hardware due to a software upgrade.

The simulator was setup to flush the caches when starting the program (i.e., at “main”, after the instrumentation package setup) to avoid situa-

tions where data was brought into the caches before the program execution starts (for instance because of the instrumentation package startup-phase touching the functions). Our accumulated values for real hardware excludes initialization and cleanup of the instrumentation library, but does not invalidate the cache contents.

The benchmarks were instrumented with four methods, source-based instrumentation (split in inlined and non-inlined operation), Dyninst (version 4.0.1 of the Dyninst API, function instrumentation with tracetool), and our LOPI framework. The source-based instrumentation was added by hand, a tedious task that required us to add instrumentation points to over 500 places for the largest benchmark (176.gcc). The 176.gcc benchmark also illustrates the effectiveness of our stub reuse, requiring only two stubs for 54 instrumented functions. For all 92 instrumentation points (in all benchmarks), totally 5 different stubs were needed.

To get comparable results, we implemented the same instrumentation for each package. The instrumentation performs a fairly common instrumentation operation, reading a 4-byte value at function entry and accumulating it at the function exit, similar for instance to accumulating a hardware performance counter (the kernel is not accessed). We exclude the perturbation caused by the OS kernel in our simulated environment by pausing the measurements on kernel entry and starting them again on kernel entry (the simulated caches are also disabled when executed kernel code). This was done to avoid timing behavior to affect the measurements and also to make the measurements more OS-independent.

5.5 Measurement results

Figure 5.7 shows the average number of instructions per function for a subset of the SPEC CPU2000 benchmarks. The length includes that of called functions (even for recursive function calls). From the figure, we can get a feeling for the cost of instrumenting functions, i.e., instrumenting a program with frequent short functions is likely to be more costly than instrumenting one with longer functions. We observe that for many ap-

Table 5.2: *Continued on next page.*

Benchmark		Cycles	Instructions	Branches nr	Branches miss pred.
164.gzip	src	1.03	1.06	1.06	1.00
	src (inline)	1.01	1.02	1.02	1.03
	LOPI	1.17	1.16	1.13	1.74
	Dyninst	1.25	1.21	1.23	1.00
176.gcc	src	1.09	1.13	1.11	1.07
	src (inline)	1.02	1.05	1.03	0.99
	LOPI	1.37	1.42	1.30	1.51
	Dyninst	n/a	n/a	n/a	n/a
181.mcf	src	1.17	1.46	1.38	1.00
	src (inline)	1.04	1.18	1.13	0.90
	LOPI	1.43	2.17	1.88	2.16
	Dyninst	1.67	2.50	2.51	1.02
183.equake	src	1.00	1.00	1.01	1.00
	src (inline)	1.00	1.00	1.00	0.99
	LOPI	1.01	1.02	1.02	1.03
	Dyninst	1.01	1.02	1.03	1.00
197.parser	src	1.03	1.07	1.06	1.02
	src (inline)	1.01	1.03	1.02	1.01
	LOPI	1.11	1.19	1.15	1.36
	Dyninst	1.21	1.24	1.25	1.03
256.bzip2	src	1.04	1.08	1.11	0.99
	src (inline)	1.02	1.04	1.04	1.00
	LOPI	1.21	1.22	1.26	2.47
	Dyninst	n/a	n/a	n/a	n/a
300.twolf	src	1.08	1.12	1.15	1.03
	src (inline)	1.01	1.05	1.04	1.01
	LOPI	1.25	1.33	1.33	1.34
	Dyninst	n/a	n/a	n/a	n/a
Average	src	1.06	1.13	1.13	1.01
	src (inline)	1.02	1.05	1.04	0.99
	LOPI	1.22	1.36	1.30	1.66
	Dyninst	1.28	1.49	1.50	1.01

plications, e.g., 164.gzip, 176.gcc and 300.twolf, a large proportion of the functions are shorter than 90 instructions (183.equake also show a large proportion of short instruction, but almost all work is done in a few long-running functions). This indicates that keeping the cost of instrumenting a function as low as possible is very important for these programs.

Table 5.2: Aggregate overhead for the SPEC benchmarks. Dyninst average values are caclulated from the successful benchmarks.

Benchmark		L1 Dcache refs	L1 Dcache misses	L1 Icache refs	L1 Icache misses	L2 unified refs	L2 unified misses
164.gzip	src	1.10	1.01	1.02	1.02	1.01	1.02
	src (inline)	1.04	1.01	0.97	0.95	1.01	0.97
	LOPI	1.29	1.04	1.12	1.06	1.03	1.20
	Dyninst	1.43	1.02	1.23	1.14	1.02	1.16
176.gcc	src	1.16	1.06	1.11	1.03	1.03	0.97
	src (inline)	1.06	1.05	1.02	1.05	1.05	0.96
	LOPI	1.54	1.32	1.46	1.13	1.14	1.08
	Dyninst	n/a	n/a	n/a	n/a	n/a	n/a
181.mcf	src	1.61	0.99	1.18	1.06	0.99	0.99
	src (inline)	1.23	1.00	1.04	1.02	1.00	1.01
	LOPI	2.62	1.14	1.43	1.65	1.00	0.99
	Dyninst	3.39	0.99	1.69	1.24	0.99	0.98
183.equake	src	1.01	1.00	1.00	1.00	1.00	1.00
	src (inline)	1.01	1.00	1.00	1.31	1.02	1.00
	LOPI	1.02	1.04	1.02	1.04	1.04	1.01
	Dyninst	1.03	1.04	1.02	1.00	1.03	1.01
197.parser	src	1.08	1.00	1.03	0.97	1.00	1.00
	src (inline)	1.03	1.00	1.01	1.01	1.00	1.00
	LOPI	1.25	1.02	1.11	1.66	1.02	1.01
	Dyninst	1.37	1.01	1.21	1.06	1.01	0.99
256.bzip2	src	1.09	1.00	1.04	1.06	1.00	1.00
	src (inline)	1.04	1.00	1.01	1.01	1.00	1.00
	LOPI	1.28	1.00	1.20	1.15	1.00	1.00
	Dyninst	n/a	n/a	n/a	n/a	n/a	n/a
300.twolf	src	1.14	1.02	1.08	1.75	1.03	0.58
	src (inline)	1.06	1.01	1.01	1.28	1.02	0.97
	LOPI	1.39	0.95	1.25	1.28	0.96	0.75
	Dyninst	n/a	n/a	n/a	n/a	n/a	n/a
Average	src	1.17	1.01	1.07	1.13	1.01	0.94
	src (inline)	1.07	1.01	1.01	1.09	1.01	0.99
	LOPI	1.48	1.07	1.23	1.28	1.03	1.00
	Dyninst	1.80	1.01	1.29	1.11	1.01	1.03

Table 5.2 provides aggregate execution times/overhead and cache behavior with source instrumentation (both inlined and not inlined), Dyninst, and the LOPI framework. We see that source instrumentation, particularly inlined, is the approach with lowest overhead (on average 13% more instructions non-inlined and 5% inlined). This is an expected result since the source instrumentation can be optimized by the compiler. LOPI and Dyninst execute 36% and 49% more instructions, respectively, than an

183.equake

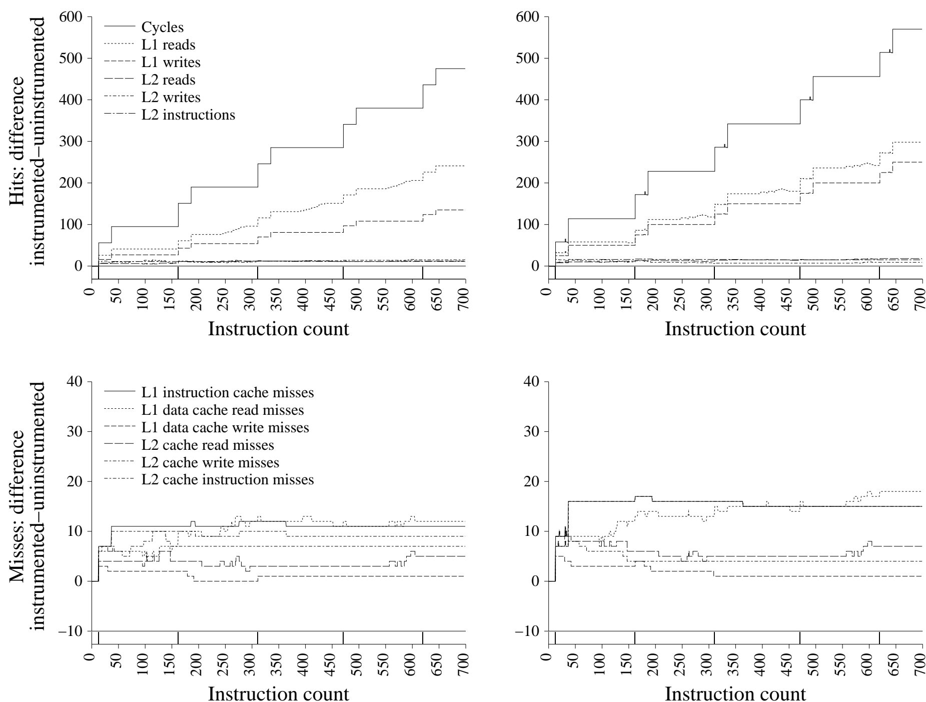


Figure 5.8: *Continued on next page.*

uninstrumented application. In terms of execution time, we find that LOPI generates 22% longer execution times on average and Dynint 28% longer execution times than an uninstrumented application.

Analyzing the cache misses we find that LOPI generates fewer first level cache accesses on average than Dyninst does, but LOPI has more first-level cache misses than Dyninst. This indicates a higher locality in the Dyninst code. However, when we look at the second-level cache accesses we find that the number of misses is comparable for LOPI and Dyninst. One reason for the higher number of data read misses for LOPI is that the return frames (which are logically code) are allocated as data.

197.parser

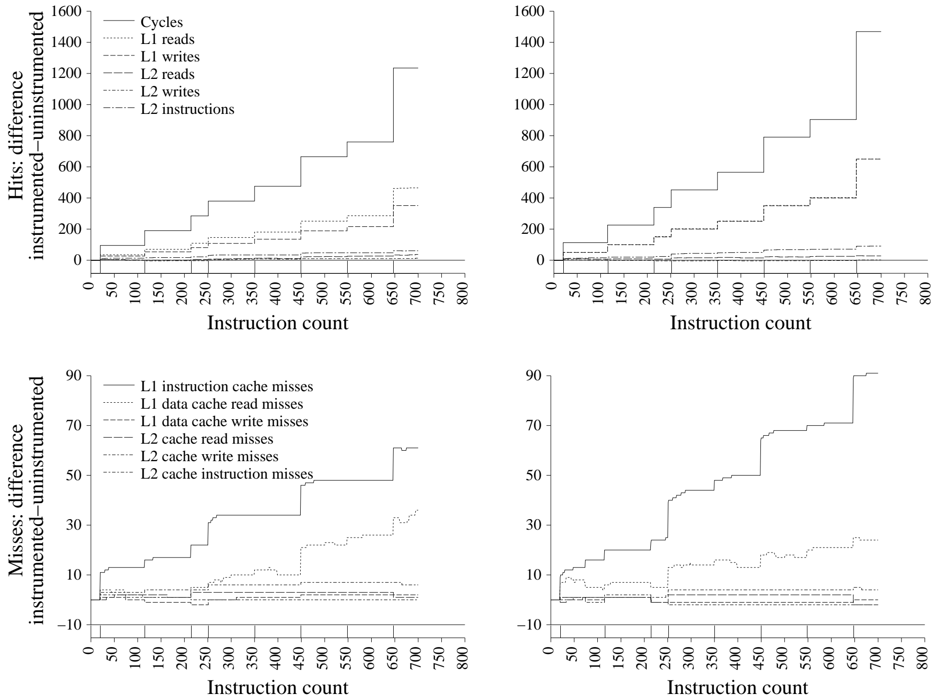


Figure 5.8: Partial execution profile for 183.equake and 197.parser. LOPI is shown on the left, Dyninst on the right.

We have identified one performance limitation for LOPI – a high number of miss-predicted branches. The Pentium III employs a branch predictor for function returns, which work as long as functions are called in the "normal" manner, i.e., through a `call`/`ret` pair. Since LOPI overwrites the return address with an address in the return frame, the return branch predictor misses its prediction, resulting in a performance loss. This problem was not visible in the simulated results.

Figure 5.8 presents a partial execution profile for the 183.equake and 197.parser SPEC benchmarks. The figure shows the difference between an instrumented and a non-instrumented run for both LOPI and Dyninst

(note that the graph does not show the absolute values, which start at higher than zero). The profiles are constructed from a trace of every instruction in the shown code snippet (except for the instrumentation code), i.e., every point in time in the figure corresponds to one instruction in the non-instrumented code. Instrumentation points for function entries are shown as vertical bars below the x-axis.

The 183.equake profile comes from the execution of a nested execution loop, which calls three short functions *phi0*, *phi1*, and *phi2* where *phi2* is instrumented. For the 197.parser profile, the instrumented section shows a section with numerous recursive function calls. As the Figure shows, the return frames cause some pressure on the caches when the frames cannot be reused on deeper levels of function nesting (because of the recursion). This is especially visible for L1 read misses that increase with each additional instrumented call in Figure 5.8.

From the graphs, we can see that the Dyninst instrumentation is more intrusive than our instrumentation. Our instrumentation is mainly cheaper when instrumenting the function returns (shown as the second climb in the upper graphs), which shows that the lazy return instrumentation pays off. We can also see that the number of cache misses is somewhat higher for Dyninst, although both instrumentation packages primarily cause cache misses on the first invocation.

5.6 Related work

In this section we discuss some other tools that are similar to our instrumentation framework. We start with those that rewrite binary files in order to instrument an application. Examples of such tools are PatchWrx [18], Etch [88], ATOM [4], and EEL [58]. We thereafter discuss Dyninst [16, 72], which rewrites the memory image in order to instrument an application.

PatchWrx, ATOM, and EEL works on RISC processors, where it is easier to rewrite and patch a binary file since all instructions have the same size. In order to patch and trace an instruction, you simply replace

the traced instruction with a branch instruction to a code snippet where the replaced instruction together with the instrumentation code reside. In contrast, rewriting a binary file for an IA-32-processor is much harder due to variable instruction length. Etch and LOPI both works for IA-32-binaries, and Dyninst is available for both RISC and CISC processors.

PatchWrx [18] is developed for Alpha processors and Windows NT. PatchWrx utilizes the PALcode on the Alpha processor to capture traces, and it can patch, i.e., instrument, Windows NT application and system binary images. PatchWrx replaces all types of branching instructions with unconditional branches to a patch section where the instrumentation code reside. PatchWrx can also trace loads and stores by replacing the load or store instruction with an unconditional branch to the instrumentation code, where also the replaced load or store resides.

ATOM [4] is developed for Alpha processors and works under Tru64 UNIX. ATOM is a general framework for building a range of program analysis tools, e.g., block counting, profiling, and cache simulation. ATOM allows a procedure call to be inserted before and after any procedure, basic block, or instruction. The user indicates where the instrumentation points are, and provides analysis routines that are called at the instrumentation points. ATOM then builds an instrumented version of the application including the analysis routines.

EEL [58] (Executable Editing Library) is a library for building tools to analyze and modify executable files. It can be used, e.g., for program profiling and tracing, cache simulation, and program optimization. EEL runs on SPARC processors under Solaris, and provides a mostly architecture- and system-independent set of operations to read, analyze and modify code in an executable file. The user can provide code snippets that can be inserted at arbitrary places in the binary code. EEL is capable of sophisticated code analysis, e.g., control-flow graph analysis and live/dead register analysis.

Etch [88] is a general-purpose tool for rewriting Win32 binaries for IA-32-processors. Etch provides a framework for handling the complexities of both the Win32 executable format as well as the IA-32 instruction set. Important issues with the Win32 format that Etch solves are to correctly

identify code and data sections, as well as identification of all dynamically loaded libraries and modules. Etch can be used, e.g., for tracing all loads and stores, measuring instruction mixes, and code transformation for performance improvements. There is also a graphical user interface provided with Etch.

Dyninst [16, 72] patches and instruments the application in-core, i.e., after the program has been loaded into memory. This approach allows instrumentation to be added to and removed from the program during runtime. For example, instrumentation can be added where new hot-spots in the code are detected during runtime, and instrumentation can be dynamically removed when it is no longer needed, which can reduce unnecessary overhead. Memory image rewriting also opens up the possibility to instrument operating system kernels [100], which cannot be restarted in order to have the instrumentation take effect.

Pin [64, 81] is a tool for dynamic instrumentation of Linux applications available for IA-32e, ARM, Itanium and IA-32e. It provides an API for inserting function calls to user-defined measurement functions at arbitrary points in the code. Pin performs the program instrumentation at run time, using a just-in time compiler to instrument and translate the application. As a result, Pin can handle shared libraries, multi-threaded applications, as well as mixed code and data.

5.7 Conclusions

Program instrumentation is an important technique in many areas, e.g., performance measurements, debugging, and coverage analysis. To be useful, instrumentation must be easy to apply and it should perturb the application execution as little as possible. In this paper we present and evaluate the LOPI framework, which provides a low-overhead generic solution to program instrumentation. The LOPI framework automatically instruments an application by rewriting the binary file(s) by adding one step in the compilation process. LOPI gives low overhead by applying

techniques to reduce the number of added instructions to the program and by using a lazy method for instrumenting function returns.

We provide detailed measurements of the instrumentation perturbation using hardware and full-system simulations of seven SPEC CPU2000 benchmarks. We compare the LOPI framework to the state-of-the-art Dyninst package and regular source-based instrumentation. The measurements show that source-based instrumentation has the lowest instruction overhead, on average 13%, but requires significantly more tedious work for instrumenting the code. Comparing LOPI and Dyninst we find that LOPI has lower instruction overhead than Dyninst, on average 36% as compared to 49%, respectively. In terms of execution time, LOPI increases the execution time by 22% compared to uninstrumented operation whereas Dyninst adds 28%.

We believe that the LOPI framework is a viable and flexible way for automatic program instrumentation with low perturbation. Future work on LOPI involves adding support for instrumentation at arbitrary program locations, which would require copying overwritten instruction into the entry stub and saving live registers at the instrumentation point. Like Dyninst does, this would require careful handling of replacing instructions, especially on architectures with variable-length instructions. Another possibility is to port the framework to other architectures than IA-32, which could require other optimizations than those explored here.

Availability

LOPI is available as free software licensed under the GNU GPL at `http://www.ipd.bth.se/ska/lopi.html`.

Bibliography

[1] L. Albertsson and P. S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 191–198. IEEE Computer Society, August 2000.

[2] AMD Corporation. *AMD Multi-Core Technology Whitepaper.* See http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf, accessed 15/7-2005.

[3] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conf. Proc.*, 30:483–485, 1967.

[4] S. Amitabh and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[5] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, Nov. 1997.

[6] J. Appavoo, M. Auslander, D. D. Silva, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, J. Xenidis, M. Stumm, B. Gamsa, R. Azimi, R. Fingas, A. Tam, and D. Tam. Enabling Scalable Performance for General Purpose Workloads on Shared Memory Multiprocessors. Technical Report RC22863 (W0307-200), IBM Research, july 2003.

[7] 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.

[8] M. J. Bach and S. J. Buroff. Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical Journal*, 63(8):1733–1749, October 1984.

[9] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*, pages 46–57, Dec. 1996.

[10] P. Bame. pmccabe. See http://parisc-linux.org/˜bame/pmccabe/, accessed 20/6-2004.

[11] 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.

[12] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating systems principles*, pages 164–177, 2003.

[13] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 2nd edition, 1998.

[14] F. Bellard. *The QEMU project homepage*, See http://fabrice.-bellard.free.fr/qemu/, accessed 4/8-2005.

[15] The Beowulf Project. *Beowulf Homepage*, See http://www.beowulf.-org/, accessed 4/8-2005.

[16] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[17] 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.

[18] J. P. Casmira, D. P. Hunter, and D. R. Kaeli. Tracing and characterization of Windows NT-based system workloads. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):6–21, December 1998.

[19] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. Technical Report TR-92-03-02, 1992.

[20] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible debugging using program instrumentation. *IEEE transactions on software engineering*, 27(8):715–727, August 2001.

[21] D. R. Cheriton and K. 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, Nov. 1994.

[22] 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.

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

[24] 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.

[25] S. Doherty, D. L. Detlefs, L. Grove, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and J. Guy L. Steele. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA '04:*

Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, pages 216–224, New York, NY, USA, 2004. ACM Press.

[26] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.

[27] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[28] B. Gamsa, O. Krieger, E. W. Parsons, and M. Stumm. Performance issues for multiprocessor operating systems. Technical Report CSRI-339, Computer Systems Research Institute, university of Toronto, November 1995.

[29] J. Garcia, J. Entrialgo, D. F. Garcia, J. L. Diaz, and F. J. Suarez. PET, a software monitoring toolkit for performance analysis of parallel embedded applications. *Journal of Systems Architecture*, 48(6-7):221–235, 2003.

[30] 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.

[31] L. G. Gerbarg. Advanced synchronization in Mac OS X: Extending UNIX to SMP and real-time. In *Proceedings of BSDCon 2002*, pages 37–45, San Francisco, California, USA, february 2002.

[32] G. H. Goble and M. 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.

[33] R. Goeckelmann, M. Schoettner, S. Frenz, and P. Schulthess. A kernel running in a DSM – design aspects of a distributed operating system. In *IEEE International Conference on Cluster Computing (CLUSTER'03)*, pages 478–482. IEEE, December 2003.

[34] 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.

[35] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. ACM, ACM, 1982.

[36] M. Greenwald and D. 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, Oct. 28–31 1996. USENIX Association.

[37] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[38] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9):901–928, 1998.

[39] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 3rd edition, 2003.

[40] 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.

[41] IBM Corporation. *PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors*, March 2005.

[42] Intel Corporation. *Intel Dual-Core Processors.* See http://www.-intel.com/technology/computing/dual-core/index.htm, accessed 15/7-2005.

[43] Intel Corporation. *Preboot Execution Environment (PXE) Specification*, September 1999. Version 2.1.

[44] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 2002.

[45] ITU-T. *ITU-T Recommendation Q.700, Introduction To ITU-T Signalling System No. 7 (SS7).* International Telecommunication Union, 1993.

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

[47] R. N. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[48] The L4Ka::pistachio microkernel white paper, May 2003. System Architecture Group, University of Karlsruhe.

[49] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance.

[50] 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.

[51] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture News Letters*, 1, June 2002.

[52] S. Kågström, H. Grahn, and L. Lundberg. Automatic low overhead program instrumentation with the LOPI framework. In *Proceedings of the 9th Workshop on Interaction between Compilers and Computer Architectures*, San Francisco, CA, USA, February 2005.

[53] S. Kågström, H. Grahn, and L. 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.

[54] S. Kågström, H. Grahn, and L. Lundberg. Scalability vs. Development Effort for Multiprocessor Operating System Kernels. Submitted for journal publication, August 2005.

[55] S. Kågström, H. Grahn, and L. Lundberg. The Design and Implementation of Multiprocessor Support for an Industrial Operating System Kernel. Submitted for journal publication, June 2005.

[56] S. Kågström, L. Lundberg, and H. Grahn. The application kernel approach - a novel approach for adding SMP support to uniprocessor operating systems. To appear in Software - Practice and Experience.

[57] 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.

[58] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the 1995 SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

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

[60] G. Lehey. Improving the FreeBSD SMP implementation - a case study. In *Asian Enterprise Open Source Conference*, Singapore, October 2003.

[61] 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.

[62] K. London, S. M. amd P. Mucci, K. Seymour, and R. Luczak. The PAPI cross-platform interface to hardware performance counters. In *Proceedings of the Department of Defense Users'Group Conference*, June 2001.

[63] R. Love. *Linux Kernel Development*. Sams, Indianapolis, Indiana, 1st edition, 2003.

[64] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[65] 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.

[66] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, 1992.

[67] 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.

[68] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. *SIGOPS Oper. Syst. Rev.*, 26(2), 1992.

[69] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read-copy update. In *Ottawa Linux Symposium*, pages 338–367, June 2002.

[70] Message Passing Interface (MPI) Forum. *The Message Passing Interface Standard*. See http://www.mpi-forum.org/, accessed 28/7-2005.

[71] Microsoft Corporation. *Windows Server 2003 Clustering*, see http://www.microsoft.com/windowsserver2003/technologies/-clustering/default.mspx, Accessed 4/8-2005.

[72] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall.

The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[73] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. *Parallel Processing Letters*, 11(2–3):267–280, 2001.

[74] MIPS Technologies. *MIPS32 Architecture for Programmers Volume III: The MIPS32 Privileged Resource Architecture*, March 2001.

[75] P. Moseley, S. Debray, and G. Andrews. Checking program profiles. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 193–203, Amsterdam, The Netherlands, September 2003.

[76] S. J. Muir. *Piglet: an operating system for network appliances.* PhD thesis, University of Pennsylvania, 2001.

[77] B. Mukherjee, K. Schwan, and P. Gopinath. A Survey of Multiprocessor Operating System Kernels. Technical Report GIT-CC-92/05, Georgia Institute of Technology, 1993.

[78] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[79] The Open Group. *Systems Management: Application Response Measurement (ARM)*, july 1998.

[80] OpenMP Architecture Review Board. *OpenMP Version 2.5 Specification*, May 2005. See http://www.openmp.org, accessed 28/7-2005.

[81] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37 2004)*, 2004.

[82] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the Linux kernel. In *Proceedings of Tools 2002*, volume 2324, pages 220–226, April 2002.

[83] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, CA, USA, February 2005.

[84] QNX Software Systems Ltd. *The QNX Neutrino Microkernel, see http://qdn.qnx.com/developers/docs/index.html*, 2005. Accessed 28/7-2005.

[85] F. 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.

[86] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, 2004.

[87] C. Robson. *Real World Research.* Blackwell publishers Inc., Oxford, 2nd edition, 2002.

[88] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. N. Bershad, and J. B. Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the 1997 USENIX Windows NT Workshop*, pages 1–8, 1997.

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

[90] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38(5):39–47, May 2005.

[91] C. Schimmel. *UNIX Systems for Modern Architectures.* Addison-Wesley, Boston, 1st edition, 1994.

[92] K. Sollins. *The TFTP Protocol (Revision 2) (RFC 1350).* MIT, July 1992. STD 33.

[93] L. Spracklen and S. 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.

[94] Standard Performance Evaluation Corporation. *SPEC CPU 2000, see http://www.spec.org*, 2000.

[95] C. Steigner and J. Wilke. Performance tuning of distributed applications with CoSMoS. In *Proceedings of the 21st international conference on distributed computing systems (ICDCS '01)*, pages 173–180, Los Alamitos, CA, 2001.

[96] C. Steigner and J. Wilke. Verstehen dynamischer programmaspekte mittels software-instrumentierung. *Softwaretechnik-Trends*, 23(2), May 2003.

[97] Sun Microsystems. Sun cluster(TM) architecture: A white paper. In *1st International Workshop on Cluster Computing (IWCC '99)*, pages 331–338, December 1999.

[98] Sun Microsystems. *Sun UltraSPARC IV*, Febuary 2004. See http://-www.sun.com/processors/UltraSPARC-IV/, accessed 28/7-2005.

[99] J. Talbot. Turning the AIX operating system into an MP-capable OS. In *Proceedings of the 1995 USENIX Annual Technical Conference*, New Orleans, Louisiana, USA, Jan. 1995.

[100] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 117–130. USENIX Association, 1999.

[101] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification*, 1995. Version 1.2.

[102] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 2004.

[103] Virtutech AB. *Simics Hindsight: Reverse execution and debugging of arbitrary virtual systems*, See http://www.virtutech.com/products/-simics-hindsight.html, accessed 4/8-2005.

[104] D. A. Wheeler. Sloccount, see http://www.dwheeler.com/-sloccount/, Accessed 20/6-2005.

[105] J. H. Wolf. Programming methods for the Pentium III processor's streaming SIMD extensions using the VTune performance enhancement environment. *Intel Technology Journal*, 3, May 1999.

[106] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, 1995.

[107] K. Yaghmour. *A practical Approach to Linux Clusters on SMP hardware*, July 2002. See http://www.opersys.com/publications.html, accessed 28/7-2005.

[108] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *11th International Conference on High-Performance Computer Architecture (HPCA-11)*, pages 266–277. IEEE Computer Society, February 2005.

[109] 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.