THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Efficient Concurrency Control for High-Performance Microkernels

Anna Lyons

Supervisor: Gernot Heiser
Assessor: Kevin Elphinstone

Thesis submitted as a requirement for the degree Bachelor of Computer Science (Honours) / Bachelor of Arts

November 15, 2011

Abstract

Concurrency control in OS kernels for multicore processor systems requires locking, which introduces complexity and overheads. One simple approach to avoiding data structure corruption is to avoid concurrency in the kernel by using one big kernel lock. Alternatively, one can identify critical sections and resources in the kernel and lock these individually, using lock ordering to prevent deadlock. Both approaches have their risks. Using many locks increases kernel complexity, risks deadlock and makes correctness hard to guarantee. But using a big kernel lock reduces scalability.

This project evaluates the giant-locking scheme of a high-performance microkernel with very short system calls, for present and near-future high-performance embedded hardware. We implement a light-weight instrumentation framework that reveals features of the system and provides detailed system traces. We investigate the scalability of a variety of workloads using simulation, and relate these to real workloads via a simple model.

Acknowledgements

Firstly, I would like to thank my supervisor Gernot for allowing me to undertake this project, even though I had yet to prove myself by succeeding in the Advanced Operating Systems course. I would like to attribute great thanks to my assessor Kevin, not for being my assessor, but for allowing me to take the time off work to focus on my thesis. You are going to enjoy a much better read as a result!

Thanks to my mentor, Aaron, for helping me every step of the way, from handling Linux faulting on global variables to showing me how to represent data in a more meaningful fashion by organising it in better charts.

Thanks to Justin, for the tissues and for reading through my results at literally the last minute, and to Tim, for reading my entire report before I had spell checked it. And thanks James, for being just as stressed as me and taking to time to give valuable opinions and help anyway.

In Swedish they have a phrase which is so much more effective than anything we use in English. “Tusen tack”. I know people use it in English too, but the Swedish syllables make it far more rewarding.

Contents

List of Figures

Chapter 1

Introduction

In recent years, multiprocessors have gained an increased popularity along-side their single-core, single-threaded predecessors. The driving force behind this change is that it has become cheaper to produce several less powerful processors that work together, rather than focus on manufacturing faster single uniprocessors. Other advantages are also present in the form of improved power efficiency, parallelism, fault tolerance and increased performance. Using multiple cores allows for increased power efficiency, as unused processors can be switched off. Multiple cores can allow for significant performance increases through designing programs to harness their parallelism [Sch94].

Nevertheless, significant improvements do not come without a cost. Operating systems (OS) for multiprocessor processors require different designs to manage multiple processors and take full advantage of the benefits available. There are many different styles of multiple processor systems, however this project will focus on operating-systems approaches for shared-memory symmetric multi-processing (SMP). SMPs are multiprocessor systems which present uniform access to shared memory for all processors. Operating system data structures are stored in this shared memory, accessible by all processors. There are two key factors to designing OSs for SMPs. First, the integrity of shared data structures must be preserved through synchronising any access to them. Second, performance must not suffer as a result of the synchronisation methods employed.

Much of the previous research into synchronisation on SMPs focuses on scalability for monolithic kernels or other very large scale systems, designed for tens and hundreds of processors. However, multiprocessors for embedded devices are already emerging, with the iPad 2 released this year having a dual-core processor.

One synchronisation approach that is generally thought to be unscalable is to place one lock around any OS operations, effectively single-threading the kernel. This approach is referred to as *giant locking*. Another approach, one that is thought to afford greater scalability, is to use many locks to protect shared resources. The amount of locking used is referred to as the *lock granularity*. The finer the lock granularity, the harder the OS becomes to design and change. System calls become more complex and correctness becomes very difficult to prove. Our hypothesis is that for high-performance, SMP microkernels (characterised by very short system calls) on tightly integrated chips, the overhead of giant locking will be small enough for a limited amount of processors.

1.1 Goals

The goal of this thesis is to investigate the scalability of a giant locking scheme on the OKL4 SMP microvisor, and establish the features of the current locking implementation. We develop a light-weight instrumentation framework in order to extract detailed system traces during a the execution of a variety of workloads over 1 – 4 processors. Finally, we use simulation to investigate scalability over more processors, the effect of lock implementation in a giant locked system, and to model a theoretical coarse-grained locking model.

1.2 Thesis Overview

Chapter 2 provides a background and broader examination of the ideas presented in the introduction, including a brief introduction to microvisors, a look at the trade-offs involved in lock-granularity, and an overview of the hardware platform used in experiments during this thesis.

Chapter 3 surveys a range of relevant related work, however it does not attempt to cover all of it as synchronisation on operating-systems is a widely explored topic. We take a look at previous SMP microkernel implementations, lock-free synchronisation approaches, applying distributed systems techniques to multiprocessor systems and previous instrumentation techniques for analysing locking.

Chapter 4 presents the overall approach and vision for this thesis project, as well as rationalising broader decisions and the relationship between different components of the implementation.

Chapter 5 takes a detailed look at the implementation of all of the systems implemented as a part of this thesis project. The chapter describes the design and implementation of an instrumentation and logging system for the OKL4 microvisor, the development of workloads for experimentation and the design of a simulator for modelling fine-grained locking.

Chapter 6 details many of the experiments run and their results, including micro-benchmarks of the OKL4 giant-lock, analyses of system traces and simulation results.

Chapter 7 presents an evaluation of the suitability for giant-locking versus a more coarse-grained approach based on the results of the experimentation and simulation.

Chapter 2

Background

The operating systems that are most commonly used in everyday life, such as Windows and Linux, have kernels that are referred to as *monolithic*. The *kernel* is the part of the OS that operates with privileged access to the processor in order to safely perform tasks that allow applications to run independently of each other. Monolithic operating systems contain all services required to run applications within the kernel. They are generally very large, complex systems that are difficult to maintain and impossible to assert correctness about.

2.1 Microkernels, Microvisors & Hypervisors

A *microkernel* attempts to minimise the amount of features in the kernel to achieve a small trusted computing base. Microkernel-based systems implement much of the operating system functionality as user-level servers run in separate address spaces. The servers communicate using interprocess communication (IPC) that is built into the microkernel and are otherwise isolated from each other and use separate resources. Microkernels have very small code bases and attempt to achieve minimal complexity.

Of particular relevance to this project is the fact the long-running, complex system calls of monolithic OSs are no longer system calls, but implemented externally by user-level servers. Examples of such system calls include file system interaction and I/O. Due to this fact, critical sections in a giant-locked microkernel are much shorter than for a giant-locked monolithic kernel. Accordingly giant locking can be expected to scale further on a microkernel.

2.1.1 OKL4

OKL4 is a third-generation, L4 [Lie95] family *microvisor* that we use for this project. A microvisor [HL10] is a microkernel that satisfies the objectives of both a microkernel and a *hypervisor*. A hypervisor is a highly efficient operating system designed to run multiple guest operating systems simulaneously, even if the guest operating systems do not support the hardware platform that the hypervisor is running on. OKL4 is distinct from previous L4 microkernels, in that it only offers asynchronous IPC in the form of *virtual IRQs*.

OKL4 is SMP-capable and can run on ARM uniprocessors and multiprocessors. Multiple versions of Linux can runs as guests on the OKL4 microvisor, which is used in generating the workloads used in this project. Each Linux runs on one *virtual CPU* (vCPU) and uses one *virtual MMU* (vMMU). A virtual CPU is effectively a single task, and a virtual MMU is the associated address space. The Linux that we run is Linux for a uniprocessor, not SMP Linux. As a result Linux's own locking scheme does not interfere with the results of this research.

For more detail on the SMP design of OKL4 and its locking strategy, see Appendix B.

2.2 Synchronisation

There are many ways, proposed and in practice, to synchronise kernel data structures for SMPs. As referred to in the introduction, one approach is to use giant locking. That is, to lock any access to the kernel with one lock. This approach effectively single-threads the kernel, as only one processor can be executing kernel code at a time. This means that system calls, interrupt requests, page faults or anything else requiring entry to the kernel must be accessed through the giant lock.

This lock can easily become a bottleneck reducing system *scalability* under *contention*. Contention of a lock occurs when more than one entity requires access to the lock at the one time. Scalability refers to how the entire system performs when adding more processors. For the purposes of this thesis, we will assess scalability as a function of cores vs. *throughput*. Throughput is dened as how much work the system can get done per unit of time. In an ideal system, adding a new processor should increase the throughput by $\frac{1}{n+1}$. The scalability of a locking system can be evaluated by examining how much throughput is added by adding new processors. Higher scalability refers to a throughput increase closer to the ideal, whilst lower scalability refers to something further away from the ideal. Figure 2.1 shows the ideal scalability curve in constrast with some more realistic scalability curves.

Alternatively, one can increase lock granularity, dividing shared data structures into small, self-

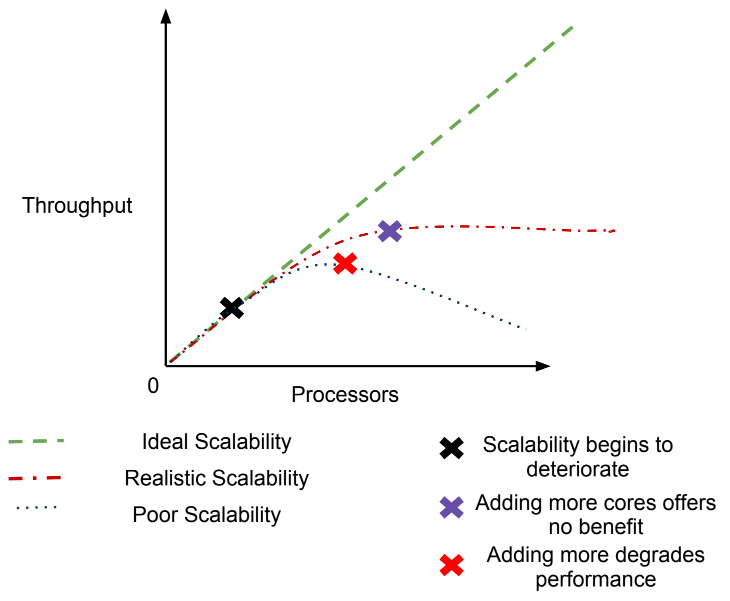


Figure 2.1: The ideal scalability curve in contrast with more realistic curves.

contained segments and critical sections that can be locked individually. Lock ordering is used to prevent deadlock. This approach, designed properly, can result in an implementation that scales much better than a single kernel lock. However designing a fine-grained locking scheme that is correct is non-trivial. If one is adapting an existing kernel, often there are many different code paths leading through separate resources and identifying critical sections becomes difficult [JAdG86].

	Giant Locking	**Fine-grained Locking**
Implementation Cost	+	-
Correctness guarantees	+	-
System call complexity	+	-
Scalability	-	+
Latency under no contention	+	-

Table 2.1: Fine-grained vs. giant locking.

As a result, giant locking cannot be discarded completely. Adding more locks increases system complexity and provides greater room for crippling problems. One runs a high risk of deadlock and each lock added increases the complexity of system calls, making the system less efficient for cases of low contention. In addition, more locks makes correctness very hard to establish.

Table 2.1 presents a quick summary of the two extremes of giant locking and very fine-grained locking. A '+' indications the advantaged scheme and a '-' indicates a disadvantaged scheme. From examining the table it can be seen that if the system scales well enough for the target hardware and kernel, giant locking would be the favoured result. However, if giant-locking does not scale well enough the benefits are irrelevant.

Lock performance is affected by critical section length. Shorter critical sections allow for greater scalability, as processes spend less time in them thus reducing contention. When using giant locking, the system calls themselves are effectively critical sections.

2.2.1 Synchronisation in the Linux Kernel

Linux 2.0 utilised giant locking. However it was quickly found that this lock became a large bottleneck, causing uniprocessor Linux to perform similar workloads more efficiently than SMP Linux on multicore processors. Performance improvements have been gained by slowly adapting the kernel to have finer lock granularity.

> In 2.2, one spin lock controlled access to the block I/O subsystem; another worked for networking, and so on. A modern kernel can contain thousands of locks, each protecting one small resource. This sort of fine-grained locking can be good for scalability; it allows each processor to work on its specific task without contending for locks used by other processors. Very few people miss the big kernel lock [CRKH05].

In general, monolithic operating systems developers, such as Linux kernel developers, have perceived that fine-grained locking implemented properly greatly outperforms coarser granularities of locks. But we must note that in the conventional monolithic systems, most services are implemented in the kernel. The implication is that a standard user load is composed of software applications that frequently make expensive system calls to I/O devices and file systems. Long system call duration means a giant lock is highly contended, as each processor must wait for other processors to leave the kernel before proceeding. As a result, throughput is limited. One can deduce this from the fact that Linux 2.0 experienced a significant performance hit for very few cores, with a significant amount of time spent waiting on locks [CRKH05].

2.2.2 Amdahl's Law

To find the maximum expected improvement to an overall system when only part of the system can be improved, Amdahl's law [Amd67] is used. When Amdahl's law is applied to parallelisation, we get the following property: the theoretical maximum speedup that can be achieved using N processors, where P is the section that can be parallelised, is:

$$\frac{1}{(1-P)+\frac{P}{N}}$$

A large part of parallel programming is spent minimising $(1-P)$.

To further rationalise why revisiting the giant-locking versus fine-grained locking debate with application to microkernels, we can argue according to an amended form of Amdal's law. Consider that, when applied to giant locking, P becomes the user time with $(1-P)$ representing time spent in the kernel. Applying fine-grained locking around shared data-structure accesses reduces the amount of time spent in the kernel and thus increases P.

However, the analysis above is missing one key factor. When considering any system one cannot simply slice a workload into P and $(1-P)$. Synchronisation techniques used to control access to the non-parallel section add more time to the original workload. So the formula used to express the actual (as opposed to maximum) speed-up that can be achieved using N processors could be expressed as:

$$\frac{1}{(1-P)+\frac{P}{N}} - f(N,C)$$

where $f(N,C)$ is the overhead added by synchronisation techniques, ie. the overhead is a function of N, the number of processors and C, the contention of the lock. But the function f itself is not so clear cut. The overhead added by a lock is a product of a number of things. First, the locking code itself adds a constant minimum overhead, which varies depending on the order of processors accessing the lock. If one processor accesses the lock over and over again, no cache operations will take place as no other processors will need to access the shared memory used in the lock. However, even at no contention, if several processors access a lock one after the other, the time taken to execute the lock code will increase due to the cache coherency operations of the hardware. Additionally, if the lock is contended, the time a processor waits in a lock is added to the locking cost.

When applied to giant-locking, we can see that $f(N,C)$ is just the cost of the one lock, increasing as more processes access the lock. Fine-grained locking seeks to minimise $(1-P)$ but at the same time increases $f(N,C)$ by including more locks. However, fine-grained locking aims to reduce contention, with processors waiting on different, less contended locks rather than one highly contended lock. As such, fine-grained locking also undesirably increases $f(N,C)$. Fine-grained locking will only succeed over giant-locking if the decrease in $(1-P)$ is signifigantly greater than the increase in $f(N,C)$.

For a monolithic kernel such as Linux it is understandable that $(1-P)$ is very large, such that fine-grained locking greatly decreases it whilst the increase in $f(N,C)$ is minimal. Additionally, due to the diversity of services offered from within Linux, one can expect that the cost of some fine-grained locks will be greatly reduced as less processes will be accessing them. For example, consider a lock protecting the sound driver. Only processes playing sound need to access this lock, thereby reducing potential cache effects and contention that would increase the cost of the lock.

However, for a microkernel the gains of fine-grained locking are far less clear. The time spent in-kernel should be a small percentage of a total workload, thus $(1-P)$ is not as large as it may be, and there are far less resources and shared data structures to divide up. As only essential services are included in the microkernel, it is far more likely that most processes will need access to most of the resources. A smaller $(1-P)$ means that it is much more likely that increases in $f(N,C)$ will come closer to the decrease in $(1-P)$.

2.3 Platform

All experiments were carried out on a NaviEngine®1 [YST+07] multiprocessor platform, using 4 ARM11™ MPCore processors with 256MiB of memory. Processor clock speed is 399 MHz and the architecture version is ARMv6. The ARM11 has an eight-stage pipeline and no L2 cache.

The benchmarking conducted in this thesis required signifigant use of the processor monitor unit (PMU) for each processor. Each PMU has one cycle counter, which can be set to tick every cycle or every 64th cycle. In addition, each PMU has two event counters, which can be set to measure cycles, or other performance characteristics including cache misses, instructions executed and branch mispredictions. The cycle counters on the PMU can only be read in privileged mode.

2.4 Summary

Previous uses of giant-locking have been on monolithic kernels with long system calls and high complexity, where giant locking was not feasible for even two processors.

The OKL4 microvisor, aimed at embedded systems provides us a platform to revisit giant- locking. We investigate that the prior criticisms of giant-locking, with the idea that they will not apply due to the unique features of OKL4 SMP and that giant-locking will provide comparable scalability to finer-grained locking, without the associated risks.

Chapter 3

Related Work

3.1 Mach

Mach [ABB+86] was one of the first microkernel implementations with a design goal of being not only multiprocessor-capable, but to offer system functions that did not limit parallelism. The major features of Mach include thread and task management, IPC, memory management, device support, multiprocessor and multicomputer support. Mach was designed to be able to run on non-uniform multiprocessor architectures as well as uniform memory architectures (or SMPs). Note that unlike L4 microkernels, the Mach microkernel kept device support and thread-management inside the kernel.

For synchronisation within the kernel, Mach used coarse-grained locking, including a lock around a shared run queue for global scheduling [Bla90].

Given Mach's poor IPC performance, even in contrast to the predecessors of L4 [Lie93], it is understandable that a giant-locking scheme would never be feasible, as the time spent in the kernel would be too long to be treated as one critical section. However, L4 microkernels offer much faster IPC performance [LES+97] and shorter system calls.

3.2 Raven: A Microkernel for SMPs

Ritchie and Neufeld [RN93] present the Raven microkernel, designed to provide a lightweight environment for multithreaded, parallel applications. Like Mach, this microkernel was targeted at SMP platforms with a small numbers of processors (up to four) and was tested on a Motorola MVME188 Hypermodule with four MC88100 RISC microprocessors. Raven is a similar microkernel to those of the L4 family, with the supervisor kernel (essentially the microkernel) providing basic abstractions such as threads and virtual memory management. One major difference is present in the fact that a user-level threading library communicates with a per-process shared-memory section within the kernel. A second major difference is that IPC is implemented at user level. Interprocess communication is implemented in the user-level kernel using shared memory to avoid the perceived performance cost of mode switching for communication.

The implementors found that using simple test&test&set [Rit93] spin locks for in-kernel synchronisation was sufficient to attain good performance. Although the synchronisation scheme is not described in great detail, they refer to using multiple spin locks to synchronise access to in-kernel critical sections in both parts of the kernel. This can be considered to be fine-grained locking. The author notes that on other hardware, cache coherency could form a bottleneck that would be amended by using more complex spin-lock based locking procedures. The hardware Raven was designed for implemented a memory bus snooping protocol to allow for cache coherency between all caches in the system, reducing contention caused by spin locks.

The Raven kernel is relevant as it is a microkernel designed for small-scale SMPs. It also shows the use of a locking scheme suitable just for a specific set of hardware using a simple style of lock rather than something more complex for scalability. However Raven appears to be just that — designed as a proof of concept at a stage when multiprocessor hardware approached a more accessible cost. Scalability is not a design factor, neither is high performance or power management. Additionally, Raven uses shared memory between the kernel and the user-level threads library rather than IPC, based on the performance of poorly implemented IPC mechanisms in Mach [ABB$^+$86]. However, IPC performance has increased dramatically with second-generation microkernels (such as L4), so such shared-memory management (which introduces security risks as well as memory management overhead) is not required.

3.3 A Lock Free Multiprocessor OS Kernel

Massalin and Pu [MP91] demonstrated that it is possible to implement SMP kernels without using locks at all. The approach in this paper implements all kernel data structures as lock-free using single and double word compare-and-swap operations. The authors adapt the Synthesis kernel [PMI88] and run it on a dual-68030 Sony NeWS workstation, claiming minimal overhead.

However it is important to note that without compare-and-swap being implemented efficiently by the hardware, this sort of approach has so far been found to be infeasible. The authors claim by using such a lock free implementation synchronisation overhead is avoided, there is increased concurrency, avoidance of deadlock. Priority inversion is also eliminated.

The benchmarks shown are only for two processor hardware, so it is impossible to tell how the implementation scales. It is possible that the bus contention created by using compare-and-swap so frequently would cause performance degradation comparable to using spin locks given more cores.

As with the Raven microkernel, this kernel was implemented as a proof of concept and not as production software. It depends on specific hardware operations and has not been designed specifically for scalability. The authors acknowledge that without native hardware compare-and-swap, locks are necessary. Applying the same methods to larger scale production software was not in the scope of the project.

3.4 Lock Free L4

Hohmuth and Härtig [HH01] approach lock-free synchronisation by developing an L4 microkernel, Fiasco, with the aim of building nonblocking real-time systems. The system incorporates locks whilst preserving the property of being wait-free. Fiasco, at the time, ran only on uniprocessors, however the authors assert that the approach lends itself well to multiprocessor-system implementations. Lock-free synchronisation is used to implement frequently-accessed global state, and wait-free techniques are used for global state that is not relevant to real-time components of the system.

Fiasco does not depend on double-word compare-and-swap, unlike the Synthesis kernel implementation described above. Instead, interrupts are disabled to perform multiword atomic updates. The authors recommend using simple spin-locks to conduct such updates with a multiprocessor. Fiasco could only be targeted at a small number of processors if it were to use such a technique.

Of particular interest is how Fiasco handles the synchronisation of thread state. Part of the thread state is protected by wait-free locks, however the IPC-critical portion of thread state uses lock-free synchronisation.

By using a minimal number of locks, Fiasco avoids the high risk of deadlock found in fine-grained locking implementations. However, a risk of deadlock definitely exists, with the described implementation including a lock count for each thread to make sure a thread does not sleep while holding a lock. The careful interleaving of lock-free data structures with wait-free locking appears to be similarly hard to get completely correct as a fine-grained locking implementation. Finally, whilst the authors claim that Fiasco would be easily adapted to and perform well on a multiprocessor system, we lack a solid implementation with benchmarks to see how well it could scale. The authors refer to research showing that lock-free approaches offer less overhead than wait-free or lock-based synchronisation techniques [ARJ97]. This research, however is only focused on uniprocessors and does not consider the expense of atomic operations on multiprocessor architectures.

3.5 Corey

Boydwickizer et al. [BWCC$^+$08] present an OS named Corey designed explicitly to reduce sharing in order to increase scalability. Specifically they allow applications to define sharing requirements for kernel data. The paper argues that the kernel should arrange data structures such that only one processor need update it, unless applications specify otherwise. They achieve communication through shared memory.

When sharing across processors is required, Corey uses MCS locks as presented by Mellor-Crummey and Scott [MCS91] with fine-grained locking. Corey was tested on Intel Xeon and AMD Opteron processors with up to sixteen cores. Benchmarks are run in comparison with Linux showing a 25% speed improvement, although the authors acknowledge that Corey is not a fully fledge monolithic OS and lacks a lot of the features that cause Linux to operate slower. However they do claim most of the performance gain is relative to the scalability enhancements made in the design of Corey.

This paper is of interest to us because it claims increased scalability based on a reduction in locking and use of fine-grained locking. Their findings will differ significantly from ours as Corey is based on a monolithic kernel, written from the ground up to achieve scalability. As far as we have read it is not aimed at embedded systems, and to apply the synchronisation and data organisation techniques in Corey would require substantial changes to OKL4.

3.6 Hierachial Clustering

Many approaches to operating system scalability propose the use of distributed systems principles in order to develop a scalable microkernel operating system. Hierachial clustering as presented in Urau et. al. [USK93] aims to produce an operating system for a broad range of memory models (non-uniform multiprocessor architectures (NUMA) as well as SMP) suitable for a high number of processors. Small groups of processors are assigned to different *clusters*, where each cluster shares kernel data structures as a small-scale, efficient multiprocessor operating system. For large systems there are multiple clusters, which communicate and collaborate through shared memory to present the interface of a single operating system.

In specific application to this thesis topic, clustering is used to reduce the amount of fined grained locking. By representing the operating system as a collection of operating systems, each for only a few processors, the authors attain a much coarser granularity of locking thereby simplifying the entire system and reducing risk of deadlock and concurrency errors. Results are presented that show that smaller clusters have lower contention, thereby reducing the stress on the coarse grained locking system and increasing kernel response time.

3.7 Multikernel

Yet another approach furthers the use of distributed system concepts so far as to treat the entire system as a distributed system. This is presented as a *multikernel* approach in Baumann et al. [BPS$^+$09, BBD$^+$09]. Message passing between kernels is used to achieve synchronisation, rather than relying on shared data. The Barrelfish OS has been designed centered around these ideas. Its design is similar to that of a microkernel, in that device drivers are user-level and message-based communication is conducted between processes to achieve protection and isolation. However unlike OKL4 SMP, each core in the machine is treated separately. No data is shared between cores apart from message channels.

This paper is relevant as it shows a structure close to that of a microkernel with high scalability. However similar to Corey, the Barrelfish is an OS built from the ground up with scalability as one of its main goals. It seeks to present and define a new OS paradigm as a way forward in a future of massively multicore systems and does not optimise for the small case of only a few processors.

3.8 Microkernel Scalability

Microkernel scalability has also been investigated, with a scalable microkernel implemented with a dynamic locking scheme and dynamic granularity of locking. The kernel was capable of adjusting its locking scheme and granularity safely at run time, based on application feedback. This research was conducted on an earlier incarnation of L4, L4Ka::Pistachio [Uhl05].

The project shows that for very high loads and amounts of processors, fine-grained locking will always scale better. However, with its use of dynamic lock granularity it acknowledges that fine-grained locking can impact performance under low contention. The project itself solves the opposite problem to that which we are trying to solve — it attempts to develop a one size fits all locking scheme that is highly adaptable. For our purposes, we want to establish the trade-offs and scalability limits of different schemes for specific hardware goals. Additionally, the designs were aimed at non-uniform memory access architectures as opposed to SMPs.

3.9 Fine-grained Locking

Fine-grained locking has been subject to criticism previously. In a paper discussing the various trade-offs involved in choosing between remote-memory access and remote invocation for kernel operations, Chaves et. al. [CDL+93] make the comment that "the impact of explicit synchronization on local operations is easy to underestimate", after finding that lock acquisition and release accounted for 49% of the total execution time in the absence of contention. The authors recommend a coarser granularity of locking, at the expense of greater potential concurrency. They note that a lock-free system could be used, however they posit that this would not offer any greater performance improvements due to the cost of atomic operations.

3.10 Kernel Instrumentation

Perl and Sites [PS96] present an example of using instrumentation to analyse an operating system. The authors used a tool called PatchRX to instrument the execution of Windows NT running some SPEC92 benchmarks and Microsoft SQL Server. They used the traces to investigate how they could speed up uniprocessor Windows NT and how they could get closer to linear scaling for multiprocessor Windows NT.

The instrumentation tool directly modified the operating system to record opcodes, timestamps and memory addresses. For multiprocessor traces, the processor number is also recorded. The log used is a 45 MiB portion of physical memory reserved at boot time. The authors were able to construct patterns of lock acquisition and measure lock contention using these techniques. Similar techniques are employed in this project, however we do not store quite as much information and use less compression.

3.11 Summary

Operating systems for SMPs is a widely explored topic, for monolithic operating systems and microkernel operating systems alike. Approaches from locking to applying distributed systems methods to avoiding locks at all by using lock-free data structures are all considered and explored heavily in the existing literature. However most research is focused on systems for a large amount of processors and for more general purpose operating systems. So far, lock-free and wait-free approaches are designed specifically for real-time systems that have only aspirations of running on an SMP system. The research conducted in this thesis applies specifically to systems targeted at a limited amount of processors and designed for tightly-coupled embedded systems. Additionally, previous research has shown that fine-grained locking on larger operating systems can cause significant overhead. Thus it is relevant and offers something new to an already diverse and detailed area.

Chapter 4

Approach

The purpose of this project is to evaluate a giant locking scheme and establish how well it scales. Given the very fast system calls of OKL4 and the tightly-coupled nature of the embedded systems it is targeted for, we believe giant locking will be a sufficient locking scheme for a larger scale than has been established on conventional, monolithic operating systems.

To show the distinct trade-offs between giant-locking and finer granularities of locking, we would take an ideal implementation of both methods and contrast their behaviour on a variety of ideal workloads. However, as we only have access to a giant-locked microvisor, and the implementation of new synchronisation schemes is a non-trivial matter — beyond the scope and time constraints of this project — we need to be slightly more inventive. Additionally, ideal workloads for multiprocessor embedded devices are a substantial research topic in themselves.

Instead, we instrument and benchmark the current locking implementation to investigate its properties and limits. We use simulation to hypothesize about hardware with more processors and a practical coarse-grained locking model. In order to run accurate simulations, we implement a logging system for the microvisor that records the details of every system call that passes through the kernel.

4.1 Design

One code design goal of this thesis was the need to be able to produce detailed system traces with minimal impact to the operation of the microvisor. The system traces would then be used as the foundation for detailed analysis into how the giant-locking scheme behaves over 1 – 4 processors. Further, the traces operate as input for a discrete-event simulator that we develop in order to simulate the workloads to run on more processors and to use the workloads as a basis for simulating a finer granularity of locking.

The work done can be divided into three parts:

Instrumentation and Logging: Develop an instrumentation and logging framework with low impact on the microvisor.

Workload benchmarking: Identifying and running workloads to produce detailed system traces for analysis and as a basis for simulation.

Simulation: Develop a simulator that could scale the system traces and model a finer granularity of locking.

4.2 Instrumentation and Logging

The goal of this part of the thesis was to be able to produce traces of all application accesses to the microvisor whilst simultaneously measuring various system properties and the length of certain operations. The following features require measuring to allow for detailed analysis:

Current processor: a unique ID of the processor that the current application is running on.

Current application: a unique ID for the current application executing a system call

System call ID: a unique id for the system call being executed.

System call entry & exit time: cost of switching between application and microvisor, as well as the system call entry and exit path.

Lock latency: cost added by the lock itself, without any waiting.

Lock-acquisition time: how long a processor waits to acquire the lock.

System call duration: the length of system calls in the microvisor.

Instrumentation overhead: any time added by the instrumentation.

Contention at time of lock acquisition: how many processors are currently trying to execute a system call.

Of these values, lock latency and system call entry & exit cost are measured using separate micro-benchmarks. Instrumentation overhead is also measured using micro-benchmarks.

In order to measure the times we store timestamps at the points shown in Figure 4.1 and compute the differences.

Note that lock latency can be divided into two parts — lock exit latency and lock entry latency. We assume that lock exit latency is a stable value, changing only due to cache operations. Therefore we do not measure lock latency whilst establishing traces, but run a separate micro-benchmark to determine the cost. This reduces the amount of timestamps that need to be stored.

Base lock-entry latency is also established using a micro-benchmark, as it cannot be distinguished from lock acquisition time during runs without too much interference in the locking code. Any increases in lock-entry latency are then factored in with lock-acquisition time.

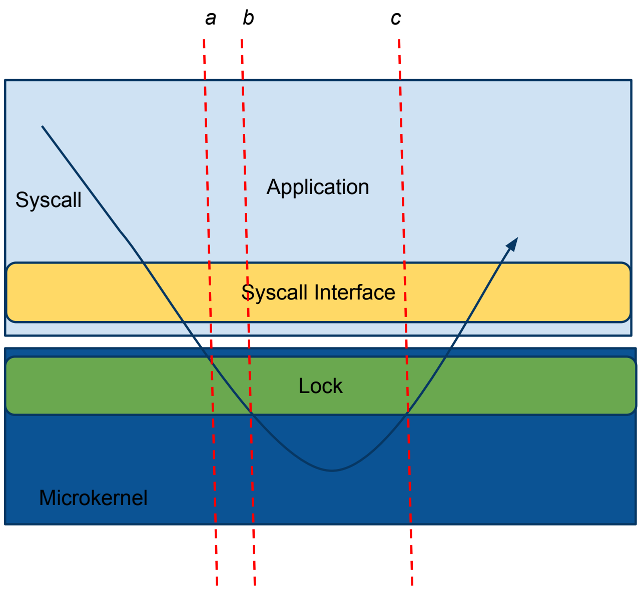


Figure 4.1: Points at which the cycle counter will be read.

4.3 Workload Benchmarking

The goal of choosing and benchmarking different workloads was to establish the limits of the giant-locking implementation and to generate variety of different traces, with different features, for input to the simulator. We chose to run all of our workloads on guest Linux instances running on the microvisor, as this provides a realistic base for an embedded device OS.

As locking scheme performance is directly related to the number and length of system calls made, we chose a variety of benchmarks that vary the system call rate.

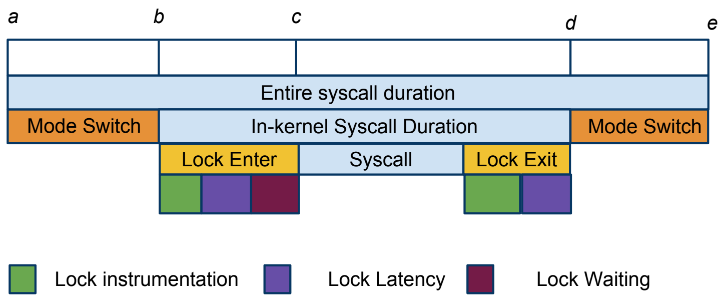


Figure 4.2: The stages of a system call with reference to the instrumentation points shown in Figure 4.1.

Benchmarking is limited by the hardware we have available, which does not have many resources. For instance, whilst the hardware does have an Ethernet port, we can currently only connect one Linux instance to it, so cannot run network-related workloads on multiple Linux instances.

We chose CPU-intensive workloads from SPEC2000 as developed by the *Standard Performance Evaluation Corporation* (SPEC). SPEC have developed a more recent suite of benchmarks, however the platform we were running the tests on only had 256 MiB of RAM, and we needed to be able to run four copies of Linux, each running two copies of the application. 50 MiB of RAM was assigned to the microvisor, as it needs to keep enough kernel stack space for each Linux instance (as well as for the serial server and the time server). This left 50 MiB of RAM for each Linux instance. As a result, we were limited to choosing applications that only required up to 20 MiB of RAM each. This is why we chose our CPU intensive workloads from SPEC2000, not SPEC2006, as even SPEC2006 benchmarks designed to be CPU intensive and not memory intensive required too much RAM.

The different workloads are listed below:

- Lmbench
- 252.eon
- 300.twolf
- A hybrid workload

Lmbench

To assess the performance of the giant-locking scheme under a high system call rate, we ran the standard Linux benchmark, lmbench [MS96], as the application for this workload. Lmbench is a series of micro-benchmarks intended to measure basic operating system and hardware system metrics. We do not present the results of lmbench, as we were using it as a workload for assessing the lock, and not as system evaluation.

300.twolf

To assess the performance of the giant-locking scheme under high CPU utilisation, we chose the SPEC2000v1.3 benchmark, 300.twolf [Swa00]. This benchmark is an implementation of the TimberWolfSC placement and global routing package that is used in the process of creating the lithography artwork needed for the production of microchips.

252.eon

We ran one other SPEC2000v1.3 benchmark, 252.eon [SCZM99] in order to make sure the results based on the runs of twolf were not particular to that SPEC benchmark. Eon is a C++ implementation of a probabilistic ray tracer.

eon-lmbench hybrid

In order to get a workload more system call intensive than the SPEC benchmarks, and less system call intensive than lmbench, we chose to run a workload with lmbench and 252.eon running side-by-side.

4.4 Simulation

The goal of this part of the thesis was to be able to vary a variety of features of the system quickly. Building a simulator allowed us to vary:

- Lock latency, to theorise about the impact of different locking implementations on scalability.
- The number of locks in the system, to analyse how our workloads would scale over different locking schemes.
- To make claims based on simulating more processors than we have available in physical hardware.

We chose to implement a *discrete event simulator* using deterministic events produced from our system traces. A discrete event simulator represents a system as a chronological order of events.

The simulator keeps track of a global system time and a list of events for each processor. Events have locks associated with them, and will only be executed by the simulator if that lock is unlocked. Initially we had thought to implement a simulator that would generate random system call events from a distribution, but we decided that this would not properly reflect the characteristics of each workload.

Chapter 5

Implementation

The previous chapter went into detail describing the overall design and vision for this thesis project, as well as rationalising design decisions and the relationship between different components of the implementation. In this section, we will dive down into the details of the implementation for each separate component.

5.1 Instrumentation & Logging

5.1.1 Log buffer

In order to store trace data, we allocated and pre-map a section of the microvisor's virtual memory region, accessible by all users, guest operating systems and the microvisor itself.

The layout of the log buffer can be seen in Figure 5.1. It is divided into four sections; a shared buffer page, a page for Linux flags, a few words of synchronisation data and the log buffer itself.

Shared buffer page

The first page is a shared page that is divided up among applications and guest OSs for the microvisor to write instrumentation values from the lock to. Each thread in the microvisor has its own address in the shared buffer to write to. This address is stored in the thread control block (TCB). Initially, for all applications and guests, this address is set to the first part of the shared buffer page. A system call has been implemented to retrieve the address of a unique, five-word section of the buffer. The system call also updates the entry in the respective TCB. By storing this address in the TCB, we avoid adding any expensive branch operations to the lock instrumentation code.

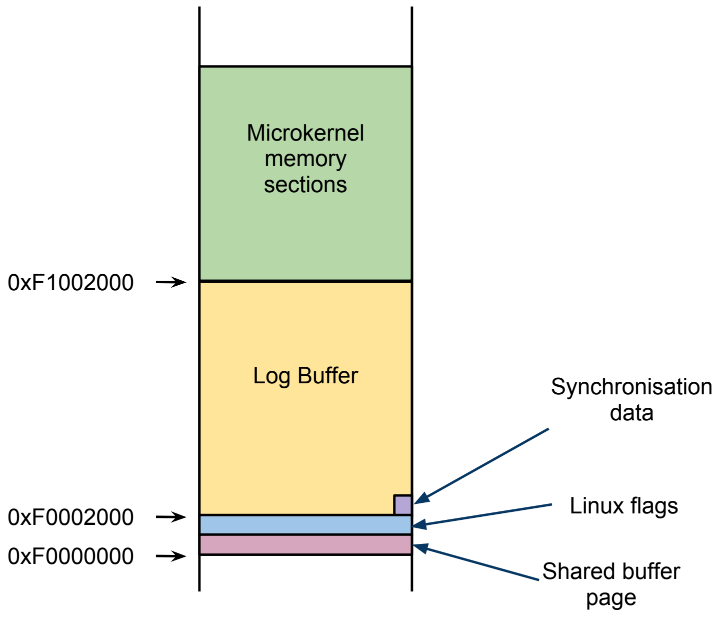


Figure 5.1: Layout of the log buffer and helper pages.

The layout of the values, as the instrumentation code writes them to a section of the shared buffer, is shown in Table 5.1.

processor ID	contention	lock entry timestamp	lock acquire timestamp	lock exit timestamp

Table 5.1: Layout of a per-thread shared buffer as written to by the lock instrumentation code.

Log buffer

The log buffer itself is simply a large region of memory that we store all of our traces in. Users interact with the log buffer using the instrumentation library described in Section 5.1.4. After every system call, an instrumentation library function is called to copy values from the shared buffer to a new entry in the log. As the microvisor build system automatically generates the system call interface, the function call is simply added during the code generation stage of the build. Other values that can be read at user time are also inserted into the log at this point, including an application ID that is specified in the XML specification for the application, and the ID of the system call that has just completed executing.

The layout of the instrumentation values in one row in the log is shown in Figure 5.2.

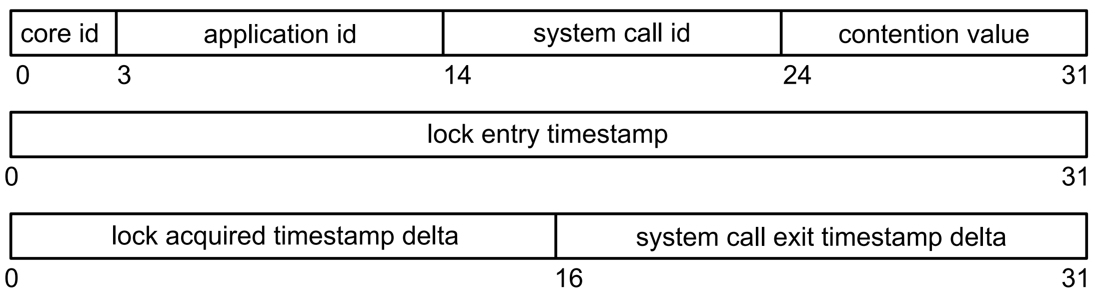


Figure 5.2: Layout of data in the 3 words that form an entry in the log buffer.

One might wonder why we do not simply write the values directly into the log from within the lock instrumentation code. We do not do this to avoid adding any extra branching operations to the lock, and also to maintain temporal locality in the addresses accessed by the locking code. Additionally, we would still require more writes to the log in order to record the user-side records. Finally, to save time in the lock we simply write all the values as full words. However our log memory is not endless, so when copying the data out we also compress it.

Synchronisation data

Since multiple applications access the log buffer, we require some atomic variables to preserve the integrity of the log. The synchronisation data section includes a one word offset that represents the next available entry in the log. Whenever an application copies data across to the log, the first thing that occus is an atomic increment of this variable.

Linux flags

This page is used for flag values for Linux instances to read. These values need to be accessed both during Linux bootstrapping as well as afterwards. Descriptions of the usage of this region can be found in Section 5.1.4.

5.1.2 Lock instrumentation

To minimise the effects of instrumentation on the lock, we narrowed down the values that we would measure in Section 4 and implemented all lock instrumentation in optimised assembly. Recall that we would measure the following: processor ID, contention, a lock entry timestamp, a lock acquire timestamp and a lock exit timestamp. Lock instrumentation reads these five values and stores them in a five-word, per-thread shared buffer. To read the five values, we instrument the locking code in three different places.

```
#if defined(INSTRUMENTATION_ENABLED)
    /* load the current thread struct*/
    GET_CURRENT_THREAD_ASM(r12)
    /* find contention */
    ASM_GET_VADDR_GLOBAL(r9, r11)
    ldr r11, [r9, #OFS_GLOBAL_NUM_CPUS_IN_KERNEL]
    /* load the shared buffer address into r6 */
    ldr r6, [r12, #OFS_THREAD_INSTRUMENTATION_SHARED_BUFFER]
    /* Read CPU ID */
    mrc p15, 0, r10, c0, c0, 5
    /* read lock entry time stamp and store in the buffer*/
    MRC p15, 0, r8, c15, c12, 2
    /* store contention */
    str r11, [r6, #4]
    /* store the cpu id */
    str r10, [r6, #16]
    /* store time stamp*/
    str r8, [r6]
#endif
    ACQUIRE_GLOBAL_LOCK_SYSCALL
#if defined(INSTRUMENTATION_ENABLED)
   /* read lock acquire timestamp */
   MRC p15, 0, r11, c15, c12, 2 /* r11 is the lock acquire cycles */
   /* store the timestamp */
   str r11, [r6, #8]
#endif
```

Figure 5.3: Lock entry instrumentation

Before entering the lock

The lock is acquired on the system call execution path using the assembly macro ACQUIRE_GLOBAL_LOCK_SYSCALL. The first instrumentation point is just before this macro is used. At this point we record the core ID, contention value and lock entry timestamp. Each value is trivial to read:

Core ID: the microvisor already stores the ID of the currently executing CPU in one of the ID registers on the coprocessor, so we read it from there and copy the value to the shared buffer.

Contention: a value for how many processors are in the kernel is also maintained by the microvisor at a specific memory address. We simply read from that address.

Lock entry timestamp: We read counter register 0 — set to increment every cycle — to capture the timestamp.

After reading, we store each value in the same order that they were read, to the shared buffer. The location of the shared buffer is stored in the current thread structure.

After acquiring the lock

The second instrumentation point is just after the lock is acquired. At this point, we read just one value:

Lock acquire time: We read counter register 0 — set to increment every cycle — to capture the timestamp.

We do not need to recalculate the address of the shared buffer as we store this in `r6`, which is not clobbered by the locking code. The assembler code for the first two instrumentation points can be seen in Figure 5.3.

```
#if defined(INSTRUMENTATION_ENABLED)
    /* read the shared buffer address */
    ldr r12, [sp, #OFS_THREAD_INSTRUMENTATION_SHARED_BUFFER]
    /* read the exit time */
    MRC p15, 0, r8, c15, c12, 2
    /* store in the buffer */
    str r8, [r12, #12]
#endif
    /* lr, r8,r9,r10,r11 will be clobbered */
    RELEASE_GLOBAL_LOCK
```

Figure 5.4: Lock exit instrumentation

Before releasing the lock

The final place we instrument the lock is just before the lock exit code is executed with the `RELEASE_GLOBAL_LOCK` assembler macro. At this point, we read just one value:

Lock exit time: We read counter register 0 — set to increment every cycle — to capture the timestamp.

At this point a full system call has taken place, so we also need to recalculate the address of the shared buffer. The lock exit assembler code can be seen in Figure 5.4

5.1.3 Microvisor system calls

We needed to implement several system calls for the microvisor to aid in benchmarking. Descriptions of each follow.

```
void _okl4_sys_instrumentation_n_cycles(okl4_word_t n);
```

A system call to wait for an amount of time based on *n*. Our benchmarks did not require exactly *n* cycles to pass, rather we just needed to spend an varying amount of time in the kernel. As a result this system call does not wait exactly *n* cycles, but executes a for loop on a volatile variable, counting down from the argument variable

```
void _okl4_sys_null_syscall(void)
```

A system call that does nothing but return to user.

```
void _okl4_sys_instrumentation_zero_counters(void)
```

A system call to zero all of the coprocessor counters (counter 0, counter 1 and the cycle counter) on all running processors. We zero the current processor counter using the PMU (recall Section 2.3, and then send an interrupt to each other processor. When a processor receives that interrupt, they handle it by zeroing their own coprocessor counters.

```
struct _okl4_sys_instrumentation_read_counters_return
_okl4_sys_instrumentation_read_counters(void)
```

A system call to read all of the PMU counters (counter 0, counter 1 and the cycle counter). The return structure is automatically generated according to the OKL4 build system semantics for system calls with multiple return values.

```
okl4_word_t _okl4_sys_instrumentation_read_cycles(void)
```

A system call to read the PMU cycle counter for the current processor.

```
okl4_word_t _okl4_sys_instrumentation_retrieve_shared_buffer(void)
```

This system call allocates a part of the shared buffer page, assigns that section to the currently executing thread and returns the address of the allocated section to the user.

5.1.4 Instrumentation library

We implemented an application level instrumentation library to manage the log buffer. Note that by application level, we mean application level with respect to the microvisor. This means that Linux applications do not have access to the instrumentation library, whilst Linux kernel instances do. As a result, the library contains a set of `#ifdefine`'s to build slightly differently for Linux. In some cases, this simply means redefining `printf` to be the Linux equivalent `printk` or `assert` to be the Linux `BUG_ON` macro. However, some functionality changes are required for Linux too. These functionality changes are outlined in the following description of the interface.

Specifically, this library handles copying from the per-application shared buffer into the log buffer. The library maintains an atomic offset that marks the next empty entry in the log buffer.

```
void instrumentation_setUpSharedBuffer(void)
```

This function uses the microvisor system call, `_okl4_sys_instrumentation_retrieve_shared_buffer`, to set up a shared buffer for the application. It also resets the current offset in the log to zero.

```
void instrumentation_logFromSharedBuffer(int syscallNumber)
```

This function is automatically called from every system call, as it is placed in the auto generated, user-level system call interface. It updates the atomic offset variable to work out which row of the log should be updated. It then writes the system call number, and the values from the shared buffer to the log.

Before performing any of this however, it checks whether a global variable representing the shared buffer has been set up and only logs data if the shared buffer address is not zero.

However, the Linux version of this function differs. Instead of checking a global variable, it checks a predetermined address from the top of the log buffer region (see Section 5.1.1). This is due to the fact that Linux makes many system calls whilst starting, at which point any global variables in the instrumentation library may not be mapped in yet. We can fix this by forcing global variables in the instrumentation library into the `.init` section, however they will then be unmapped when then mappings for the `.init` section are flushed. In order to get around this whole problem, we simply use a part of our pre-mapped log instead.

`void instrumentation_reset(void)`

This function atomically resets the offset into the log buffer to 0.

`void instrumentation_dumpLogToSerial(void)`

This is the first of two versions of this function, as both the prototype and behaviour differ wildly if this is a normal application or a Linux instance.

This version of the function simply prints out the contents of the entire log buffer, effectively dumping it to the serial port.

`unsigned long instrumentation_dumpLogToSerial(unsigned long *addr); (Linux Version)`

Using the serial port on our platform turned out to be a very slow way of extracting the data. Therefore, the Linux version of this function copies the entire log buffer across to user memory at `addr`, using Linux copy-out semantics. The user can then extract the buffer in anyway they desire. In our case, we connected to Linux via telnet over an ethernet port, used this function to copy the log buffer over, and then printed the buffer out on the telnet terminal.

5.1.5 Linux system calls

Linux applications do not have access to system calls provided by the microvisor. As mentioned previously, only Linux kernel instances have access to the instrumentation library. As a result, we needed to implement system calls to perform either of these functions. The following describes each system call we implemented and used.

`void sys_zero_counters(void)`

This system call delegates to the microvisor, calling `_okl4_sys_instrumentation_zero_counters`.

`unsigned long sys_read_cycle_counter(void)`

This system call delegates to the microvisor, calling `_okl4_sys_instrumentation_read_cycles`.

`void sys_register_shared_buffer(void)`

This system call delegates to the instrumentation library, calling `instrumentation_setUpSharedBuffer`.

`unsigned long sys_dump_buffer(unsigned long *addr)`

This system call delegates to the instrumentation library, calling `instrumentation_dumpLogToSerial`.

5.1.6 Limitations of the logging implementation

One important part of the OKL4 microvisor lock implementation is that the Linux instances, serial server and timer server are not constantly on the same physical processor, and may change processor while waiting on the giant lock. Due to this fact, the lock timestamps are not completely precise. Consider the following situation:

1. a vCPU enters the lock on physical CPU 1.
2. The lock instrumentation records a lock entry timestamp using the cycle counter for CPU 1.
3. Whilst waiting on the lock, the vCPU is migrated to CPU 2.
4. After executing acquiring the lock and executing the system call, the vCPU reads the cycle counter on CPU 2.

The result is that the delta values we store for lock acquisition time and system call duration are occasionally read from different CPU cycle counters. Since the cycle counters are all zeroed at approximately the same time, the values should not be more than a few cycles out.

This problem could be solved by using the global cycle counter available on the snoop control unit (SCU), however this would increase the impact of lock instrumentation.

5.2 Simulator

As discussed in the previous chapter, in order to model coarse grained locking, scalability for over four processors, and to vary the lock latency, we have implemented a discrete event simulator. The simulator is designed to replay the traces recorded using the logging system.

The simulator is implemented in Java, in order to harness Java's convenient BigInteger library and benefit from the associated development speed improvements that come with using a higher level language.

5.2.1 Implementation

The simulator operates on a processor abstraction, mapping all of the events from one real processor to multiple simulated processors.

Initial simulations were based on assigning application IDs to each processor to duplicate, to reliably add more workloads and to avoid increasing the system calls added to the logs by the timer-server and serial server. However, as timer-server and serial-server events would also increase with more Linux instances, we found that by mapping simulated processors to real processors a more realistic simulation was achieved.

The simulator has two stages: queue-building and simulation. Additionally, the simulator is configured by a configuration in JavaScript object notation (JSON) format, which is easily serialisable to Java objects.

Queue-building

In the queue-building stage, the simulator builds the event queues for each simulated processor from the input log file. One row from the log is given to a simulated processor if the following is true:

$simulated\_processor\_id \bmod num\_actual\_cores == actual\_processor\_id$

For each row from the log, we add a chain of events to each applicable simulated processor queue. Events are given a duration in cycles. The exact events for each row are added according to the *system call scheme*. A system call scheme simply refers to the events that are added for each system call. We implement two different system call schemes; a giant-locked scheme and a coarse-locked scheme. The duration of each event type comes from different sources. An idle event is also added at the start for each processor, with the duration being the first event for that processor, minus the first log row timestamp.

Algorithm 1 Idle time calculation

```
$prevLockExit \leftarrow prevLockEntryTimestamp + prevSyscallDuration + prevLockAcquire$
$idleDuration \leftarrow lockEntryTimestamp - prevLockExit$
```

Giant-locked system call scheme

The chain of events that is added to a simulated processor queue for each row is presented in Table 5.2. The same event chain is added for each row in the log, regardless of the system call number.

Event name	Duration source
Mode switch	Configurable
Lock entry	Configurable
System call	Duration is taken from the trace file
Lock exit	Configurable or taken from a hard-coded distribution, its generation is described in Section 6.1.5
Mode switch	Configurable
Idle	Calculated from the log - see Algorithm 1.

Table 5.2: Events generated for a row in a trace file for a giant-locked system call scheme.

Coarse-grained system call scheme

The coarse-grained system call scheme is more complicated than the giant-locked scheme as we use several different locks and exclude some system calls from locking at all. System calls are divided into two different categories: locked system calls and unlocked system calls. Table 5.3 shows the events added for a trace file for unlocked system calls.

Event name	Duration source
Mode switch	Configurable
System call	Duration is taken from the trace file
Mode switch	Configurable
Idle	Calculated from the log - see Algorithm 1.

Table 5.3: Events generated for a row in a trace file for an unlocked system call in a coarse-grained system call scheme.

Locked system calls generate many more events. To model coarse grained locking, we add multiple locks and divide up the system call duration into several parts. A sample set of events generated is shown in Table 5.4. Further detail of the exact coarse-grained locking model used is described later in Section 6.4.4.

Simulation

Following the queue building phase, the simulator starts the actual simulation. The simulator keeps track of a list of current events — one for each processor that has not yet finished. For every pass of the simulator, we choose the event that will finish soonest. An event is only executable if it is not blocked by a lock. We then subtract the duration of that event from every other executable event in the

Event Name	**Duration source**
Mode switch	Configurable
Lock entry	Configurable
System call	x% of duration taken from the trace file
Lock exit	Configurable or taken from a hard-coded distribution, its generation is described in Section 6.1.5
System call	y% of duration taken from the trace file
Lock entry	Configurable
System call	z% of duration taken from the trace file
Lock exit	Configurable or taken from a hard-coded distribution, its generation is described in Section 6.1.5
Mode switch	Configurable
Idle	Calculated from the log - see Algorithm 1.

Table 5.4: Events generated for a row in a trace file for a locked system call in a coarse-grained system call scheme. Note that the locks being acquired and released are different locks.

current execution queue, simulating their execution of that much time. Finally, we update our current events list to make sure we still have one event for each processor.

The simulator algorithm is presented in Algorithm 2 and Figure 5.5 shows an illustration of how simulator events proceed. The implementation has one nuance; even though we pick the shortest executable event from the current events list, it is possible that the event we chose was an unlock event. After executing the shortest executable event, we will potentially unblock a shorter event. Note that the only event that can be unblocked is a lock event, so the simulator does not need to recurse, as a lock is always longer than an unlock.

The simulation ends when all processors empty their event queues.

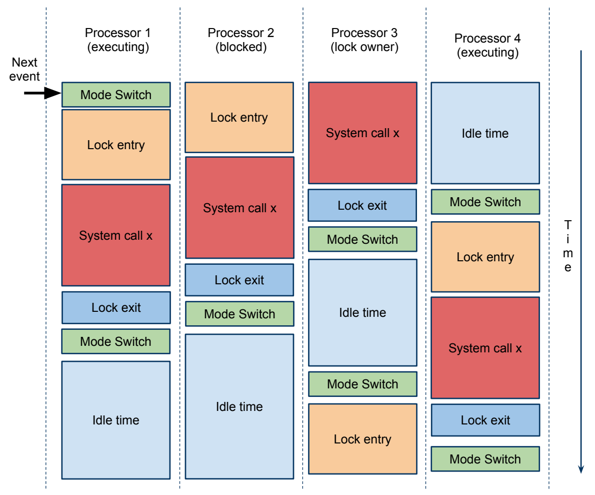
Configuration

The simulator has several configuration options:

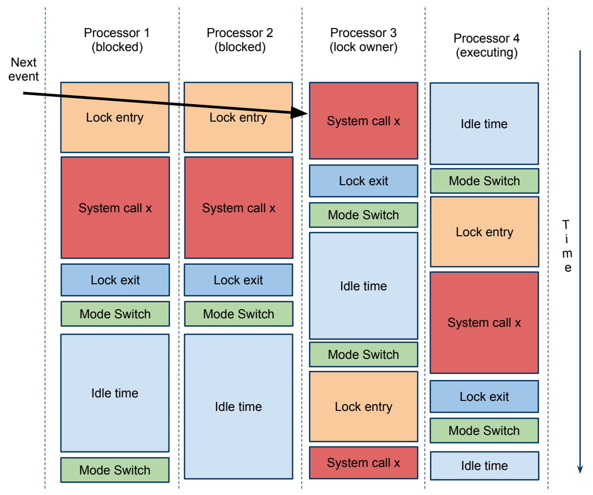
simulatedCores The number of processors we are simulating

logFile The log file to take trace data from

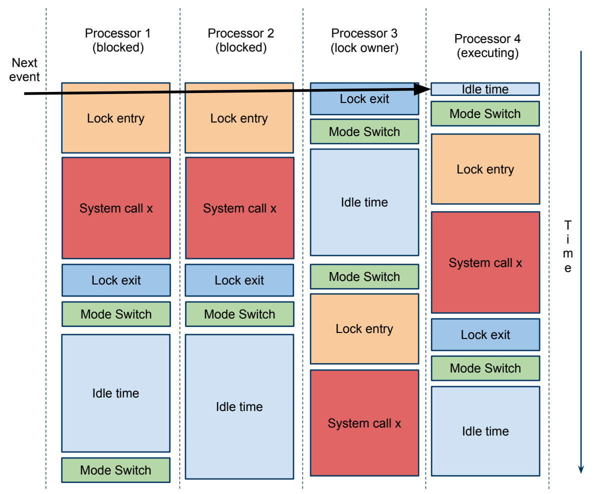
defaultLatency The default lock latency in cycles. If isDefault is true we will use this value as the lock-entry latency, and use the hardcoded distribution for lock exit latency. Otherwise, we will split this latency into $\frac{1}{3}$ for lock exit latency and $\frac{2}{3}$ for lock entry latency.



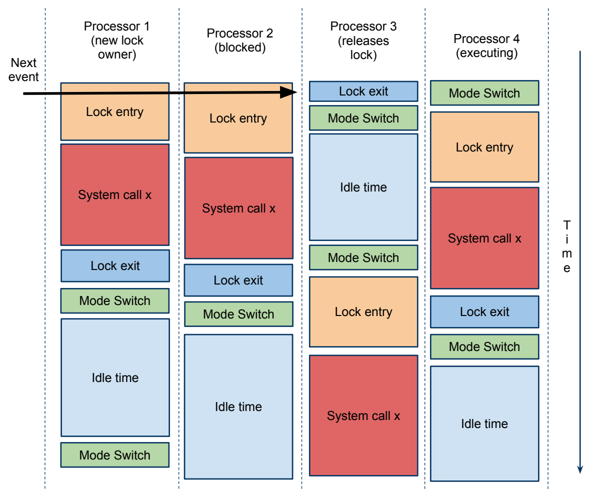
(a) Stage 1: 1 processor blocked, one process holding the lock.



(b) Stage 2: 2 processors blocked.



(c) Stage 3: 2 processors blocked.



(d) Stage 4: Lock holder changes, 1 processor blocked.

Figure 5.5: A sample execution of the simulator, showing a replay of a 4 processor trace.

Algorithm 2 Simulator event execution cycle

```
$soonest \leftarrow currentEvents.removeShortestExecutableEvent()$
$timeslice \leftarrow soonest.getDuration()$
$soonest.execute(timeslice)$
for all $currentEvents$ do
  if $event.canExecute()$ then
    $remaining \leftarrow event.execute(timeslice)$
    while $remaining > 0$ do
      $nextEvent \leftarrow event.getProcessor().getNextEvent()$
      $remaining \leftarrow nextEvent.execute(remaining)$
    end while
    if $nextEvent.isFinished()$ then
      $nextEvents.add(nextEvent.getProcessor().getNextEvent())$
    else
      $nextEvents.add(nextEvent)$
    end if
  end if
end for
$nextEvents.add(soonest.getProcessor().getNextEvent())$
$currentEvents \leftarrow nextEvents$
$time \leftarrow time + timeslice$
```

modeSwitchCost The mode switch cost to use in simulation (in cycles).

isDefault True if we are using the default lock-exit latency distribution, false otherwise.

isCoarseGrained True if we are using a coarse-grained system call scheme, false otherwise.

actualCores The number of processors in the log file.

These options are configured by using an input file. A sample is shown in Figure 5.6.

5.2.2 Limitations

The simulator is accurate for replaying events, but is just designed to be an estimation tool. The following list enumerates important features that the simulator does not implement:

- Hardware effects of increasing the number of processors (cache coherency protocols, bus contention).
- Increased scheduler activity and load.

```
{
    "simulatedCores":3,
    "logFile":"lmbench.3.sim.in",
    "defaultLantency":179,
    "modeSwitchCost":39,
    "isDefault":true,
    "isCoarseGrained":true,
    "actualCores":3
}
```

Figure 5.6: Sample simulator configuration file

- Increased time taken to execute system calls and locking code due to data structure features of the microvisor and loss of temporal and spatial locality of code execution. This is explained further in Section 6.3.
- Events will only play in the order that the simulator finds them. For instance, if a number of system calls are generated as a response to a timer interrupt, these may no longer occur at the same time in the simulator. This is due to the fact that by design, the simulator will have different times for lock waiting due to higher or lower contention. Therefore, these events may no longer execute simultaneously.

As a result, the simulator provides results that are too optimistic when it comes to the scale experiments.

Another limitation is simulation size. By loading all of the events into the simulator before simulation, we achieve a fast, but memory-intensive simulator. If we wanted to run long simulations for large numbers of processors on very large workloads, the simulator would need to be altered to build the queues as the simulation takes place. This would make the simulator more disk intensive and therefore slower, but at least it would not run out of memory. However, the current simulator bounds were enough for our desired simulations.

Chapter 6

Experimentation, Results & Analysis

In the process of this thesis many experiments were conducted to determine lock scalability, system call statistics and establish traces of various loads. The following section details each experiment and the results achieved as well as what the implications of those results are.

Experiments were conducted on a 32-bit ARMv6 platform, as described in Section 2.3. Further experiments were run on the simulator described in Section 5.2.

6.1 Micro-benchmarks

The first stage of experimentation involved running some micro-benchmarks to establish the features of the hardware, microvisor and lock implementation.

6.1.1 System call entry & exit cost

The first experiment we undertook was to measure the system call entry & exit trip cost without any locking or system call execution time, on just one processor with one application running. The system call entry and exit path involves the stages shown in Table 6.1.

As each of these stages is very small, we did not measure them individually. The test we ran took the following form:

1. Execute a system call to zero the cycle counter.
2. Execute 10,000 empty system calls
3. Execute a system call to read the cycle counter.

The user side system call entry path.
A mode switch from user to kernel.
The kernel side system call entry path.
Executing an empty function.
The kernel side system call exit path.
A mode switch from kernel to user.
The user side system call exit path.

Table 6.1: System call entry and exit path stages.

We ran this test 50 times and excluded the first run to avoid suffering cache effects that could distort the results. The cycle counter was set to increment every 64^{th} cycle to avoid overflow. Each test measured the system call entry and exit path 10,000 times with one additional exit and one additional entry for dealing with the cycle counters. The test application was the only application active during the test. The results and result calculation can be seen on Table 6.2.

Column	**Macrocycles**	**Instructions**
Average time to execute 10,001 empty system calls with cycle counter ticking every 64^{th} cycle		
Average	12, 205	250,222
Std. dev.	10.58	131.87
*Time to execute one empty system call (above values * 64 / 10,001).*		
Average	78	25
Std. dev.	0.07	0.01

Table 6.2: System call round trip costs on a single processor for an empty system call.

Based on these results, a system call entry and exit cost of 78 cycles was used as input for the simulator.

6.1.2 Lock latency

The purpose of the next experiment was to determine a lower bound for lock latency, that is, the overhead added by using the lock in execution environments with low contention. To do this, we took an execution environment with no possibility of contention at all, by building the microvisor with the lock enabled whilst only using one processor. We then executed the same experiment as above. In this case, the system call entry and exit path involves the stages shown in Table 6.3.

The results and result calculation can be seen in Table 6.4.

The user side system call entry path.
A mode switch from user to kernel.
The kernel side system call entry path.
Acquire the global lock.
Executing an empty function.
The kernel side system call exit path.
Release the global lock.
A mode switch from kernel to user.
The user side system call exit path.

Table 6.3: System call entry and exit path stages with the giant-lock in place.

Column	**Cycles**	**Instructions**
Average time to execute 10,001 empty system calls, with cycle counter ticking every 64^{th} *cycle*		
Average	51284	940527
Std. dev.	10	144
*Time to execute one empty system call (above values * 64 / 10,001).*		
Average	328	94
Std. dev.	0.06	0.01
Overhead compared to Table 6.2		
Overhead	250 (420%)	69 (376%)

Table 6.4: System call round trip costs on a single processor for an empty system call, with the lock, without instrumentation.

Taking the difference of the results from the previously established round trip costs, the lock latency of the current OKL4 microvisor is $\sim$ 69 instructions and $\sim$ 250 cycles. This is an ultimate lower bound on the lock latency, as we are running with a hot cache in a zero-contention environment over multiple processors. Cache operations such as cache line migrations would see the lock latency cost increase.

A 420% increase in the system-call entry & exit path due to the lock seems high. The conclusion here is that the microvisor lock implementation could be revisited and further optimised. However, later results will show that a lock latency this high, or indeed higher, does not have a great effect on scalability.

6.1.3 Instrumentation effects on one processor

The next micro-benchmark we conducted had the purpose of measuring the effects of the lock instrumentation on the lock performance; chiefly, we wanted to make sure we had not increased the base lock latency too much by causing any extra cache-line refills or TLB misses.

The first part of this benchmark was conducted with one processor, with the goal of finding the overhead added to the base lock latency by the instrumentation. To measure the impact we took the average of the time to execute 10,000 empty system calls 49 times (again we ignored the result of the first run to avoid cache effects distorting the results), with and without instrumentation.

Column	**Cycles**	**Instructions**
Time to execute one null system call without instrumentation		
Average	328	94
Std. dev.	0.06	0.01
Time to execute one null system call with instrumentation		
Average	399	108
Std. dev.	0.06	0.01
Overhead		
Overhead	71 (21%)	14 (15%)

Table 6.5: Lock instrumentation effects on one processor.

The results can be seen in Table 6.5. We can see that by adding approximately 12 instructions, a $\sim 22\%$ increase in lock latency occurred. To find out what was causing the extra latency we used the other counters in the PMU to count any relevant events. For each PMU reading, the results of 49 runs of 10,001 empty system calls are shown (including the added system call for the exit path after zeroing the counters and the exit path after reading the counters). Two counters were read at a time, as the ARM11 PMU offers two event counters.

The average results of PMU analysis can be see in Table 6.6. The main reason for the overhead is poor pipeline performance. We see a large increase in instruction cache misses and data microTLB misses, however looking at the actual figures the increases are insignificant (i.e. there are three extra data microTLB misses over 10,000 runs). Additionally, it appears that the longer code allowed for a decrease in branch miss-predictions, reducing the overall impact of the instrumentation.

From these results we can conclude that our lock instrumentation is minimal and sufficiently light-weight, as it does not add any extra expensive cache misses.

	Without instrumentation		With instrumentation		
Value Measured	**Average**	**Stdev.**	**Average**	**Stdev.**	**Overhead**
Cycles	3281927	430	3992726	625	**21.66%**
Instructions executed	120044	10.44	140053	8.86	**16.67%**
Folded instructions executed	9998.20	0.37	9998.27	0.41	0%
Instruction cache miss	3.73	2.82	5.35	3.28	43.17%
Stall due to data dependency	1100462	111	1240530	81	**12.73%**
Instruction microTLB miss	0	0	0	0	0
Data microTLB miss	1.65	0.48	4.14	1.11	150.62%
Branch instruction executed	130070	19.80	130076	16.67	0.01%
Branch not predicted	30029	9.1	20026	8.30	-33.31%
Branch misprediction	1	0	1.06	1.00	0%
Data cache read access	270136	36.16	310158	29.71	**14.82%**
Data cache read miss	0.020	0.72	0.02	0.14	0%
Data cache write access	129944	10.44	140053	8.86	**16.67%**
Data cache write miss	0	0	0	0	0
Data cache line eviction	0.20	0.02	0.02	0.14	0%
Main TLB miss	0.28	0.45	0.33	0.47	14.29%
External memory request	8	1.97	9	1.36	11.96%
Stall due to LSU being full	6.77	2.11	7.84	1.48	15.66%
Store buffer drained due to CP15 operations or LSU ordering constraints	40012	2.45	40014	1.96	0.00%
Buffer write merged in a store buffer slot	0	0	20002	0	N/A

Table 6.6: Relevant PMU counter values for 10,000 empty system calls averaged over 49 runs on 1 processor. Bolded values show significant overheads.

6.1.4 Lock latency and instrumentation effects over multiple processors

We suspected that as we increased the number of processors we would observe instrumentation effects decreasing as contention and cache operations became more significant. To confirm this idea, we ran the same test as above over 2–4 processors. The results are presented in a slightly different format, as it does not make sense to measure the time to execute one empty system call with contention. Instead, we compare the total time to execute 10,001 system calls per processor, with and without instrumentation.

Processors	Average cycles overhead
1	21.66%
2	-1.93%
3	7.57%
4	2.59%

Table 6.7: Average overheads of lock instrumentation for multiple processors.

The full results can be seen in Appendix A and are presented for each processor in a test run. An averaged summary is shown in Table 6.7. For two processors, the instrumentation actually has a negative impact, presumably as the small increase in lock time causes the two programs to take the lock in an alternating fashion, thereby reducing contention. This would never happen under real workloads, as it is an artifact of running under continuous contention.

The instrumentation overhead for three and four processes decreases significantly, which is the expected result. Further, this means that the 22% increase in cycles observed on one processor is an upper bound for lock instrumentation overhead, and that the instrumentation on real, less contended workloads will be minimal. This is because the slight increase in lock latency is rendered insignificant by lock-waiting time due to contention.

6.1.5 Lock-exit latency

The final micro-benchmark involved measuring the lock-exit latency. The purpose of this experiment was to divide the lock latency cost measured earlier into two parts; lock-acquisition latency and lock-release latency such that we can subtract lock-exit latency from our earlier results. Additionally, by establishing a distribution of lock-exit latency costs for input to the simulator, we found that we could minimise lock instrumentation impact by avoiding taking a timestamp after lock exit. We can do this as lock-exit latency is expected to be constant, apart from cache operations.

This experiment involved instrumenting the system call exit path to read the cycle counter before and after the lock exit assembler code was executed, and copying the values read into the shared buffer. No other instrumentation was present. We read the cycle counter value just before and just after executing the lock exit macro, and only stored the values after reading the cycle counter for the second time, as shown in Figure 6.1. As it only takes one cycle for a coprocessor access, the values read for lock-exit latency should be accurate within one or two cycles.

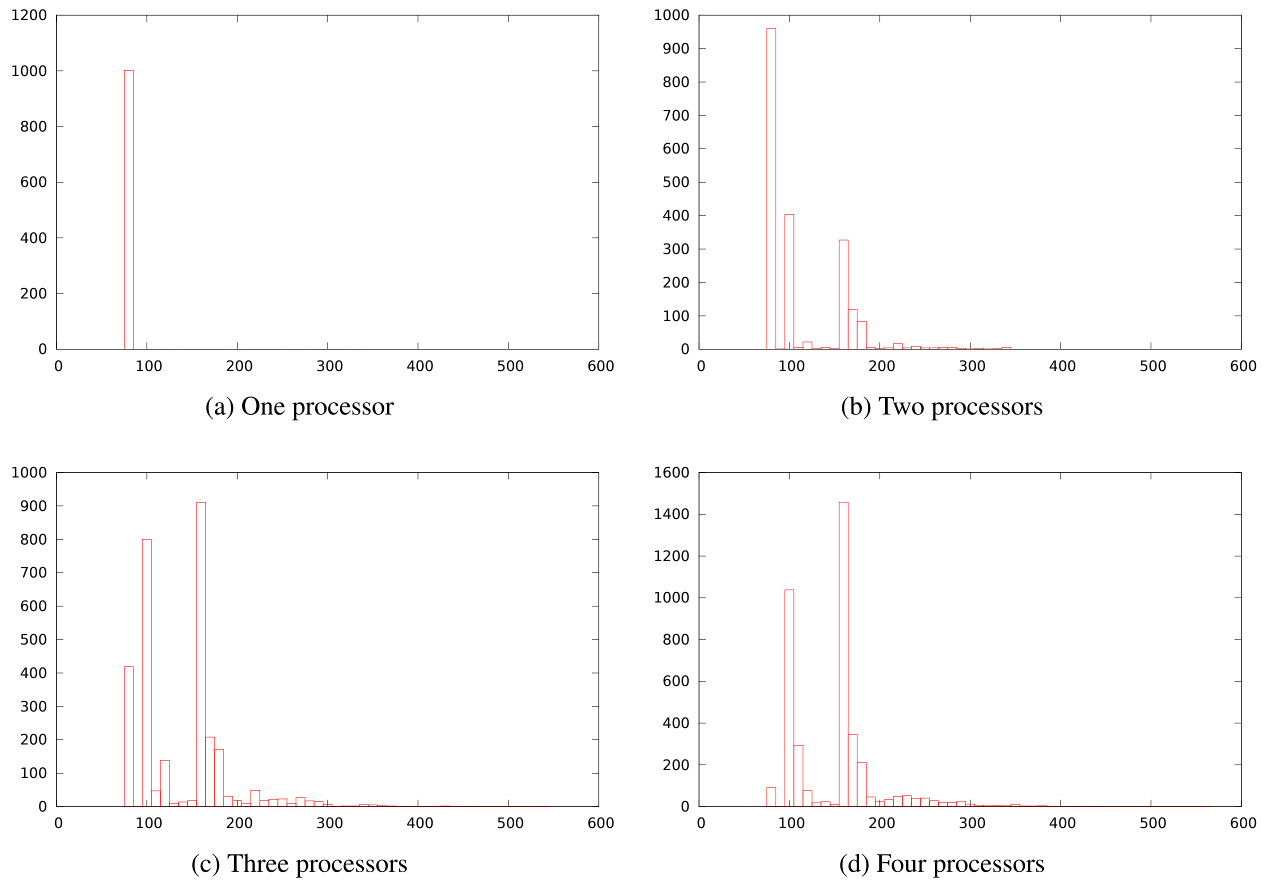
```
#if defined(INSTRUMENTATION_ENABLED)
    /* read the exit time */
    MRC p15, 0, r12, c15, c12, 2
#endif

    /* lr, r8,r9,r10,r11 will be clobbered */
    RELEASE_GLOBAL_LOCK

#if defined(INSTRUMENTATION_ENABLED)
    /* read the finish exit time */
    MRC p15, 0, r9, c15, c12, 2

    /* read the shared buffer address */
    ldr r10, [sp, #OFS_THREAD_INSTRUMENTATION_SHARED_BUFFER]

    /* store in the buffer */
    str r12, [r10, #0]
    str r9, [r10, #4]
#endif
```

Figure 6.1: Lock exit instrumentation for measuring lock exit cost only

To run the experiment, we ran the following 1000 times over 1 – 4 processors:

- Spin for a random amount of time in the user application.
- Execute the `_okl4_sys_instrumentation_n_cycles`, with a random value.

After each test, we extracted the lock-exit latency values from the traces using the instrumentation library. Histograms of the results are shown in Figure 6.2. Lock-exit latency is between 80–90 cycles when uncontended, but increases dramatically as we add more processors. For multiple processors the latency appears to be bimodal - presumably being lower when one processor takes the lock multiple



(a) One processor (b) Two processors

(c) Three processors (d) Four processors

Figure 6.2: Lock exit latency on a highly contended lock, over 1 – 4 processors. Each processor completed 1000 system calls. Axes are frequency vs. cycles.

times in a row, and higher when different processors take the lock. This is because when the same processor takes the lock sequentially, lock data is in the local cache and does not require a migration or refill.

We use these lock-exit latency values as a distribution input for the simulator in later experiments. The dramatic increase in lock-exit latency due to cache operations foreshadows later results in the report, and are the first in the theme that in general, cache contention is a much more severe problem than contention of the giant-lock.

6.2 Macro-benchmarks

We ran a series of macro-benchmarks using multiple Linux instances as guests running on the microvisor. Each individual macro-benchmark involved many different tests using a unique workload. We will refer to them as *test-sets*.

The purpose of the macro-benchmarks was to provide data as the basis for simulation, modelling and analysis.

For every test in a test-set we ran a script that interacted with all *n* Linux shells simultaneously using a python program. Each test involved the following:

- A workload running on each Linux instance. Each workload is formed of two context switching applications. By incorporating Linux context switches, we come closer to modelling a real embedded system, as a real embedded system running Linux is unlikely to be only running one application at a time.
- A serial server, for delivering output to the console and interacting with the Linux shells.
- A timer server, delivering a 100 MHz timer interrupt to each Linux instance. This interrupt serves as the end of a Linux timeslice, and signals a Linux context switch. Note that these timer interrupts are not offset in any way, so every Linux instance will context switch at approximately the same time.

For each test-set, we performed the following:

Establish a baseline: Run the test without instrumentation. Zero the cycle counter before the test and read the cycle counter after the test to obtain a baseline for a completely uninstrumented system running this workload (repeated 10 times).

Establish instrumentation effects on the microvisor: Run the test with only microvisor-side instrumentation. Zero the cycle counter before the test and read the cycle counter after the test to obtain a benchmark for a kernel-side only instrumented system. This allowed us to exclusively measure the effect of lock-instrumentation on the workload (repeated 10 times).

Fully instrumented over 1–4 processors: Run the test with kernel-side instrumentation and user-side logging to record a trace. Zero the cycle counter before the test and read the cycle counter after the test to obtain a duration to compare with the previous two results, to establish how much logging is slowing down the workload (repeated 4 times, as trace extraction was slow).

By taking three measurements to determine instrumentation effects, we can isolate how much we slow the microvisor down in contrast to how much we slow the user processes down. Recall that microvisor-only instrumentation means that we are running with lock instrumentation only. When we add the full instrumentation, this means that we call an extra function after every system call in order to copy the values from the shared buffer to a row in the global logs. Note that we only measure the cost of instrumentation when running on one processor. This is because the effects are relatively small (less than 1%) on one processor, and as we can see from the instrumentation effect micro-benchmarks, these will only diminish as we add more processors.

To extract the log data from memory, we connected to the first Linux instance over Ethernet and executed a small program that would use the `sys_dump_buffer` system call to copy the logs into Linux user-space memory and print them out over the fast ethernet connection.

6.2.1 Limitations

Time Limits

For the SPEC2000 tests, we did not run with the full reference data sets. This is because, for each test set, we needed to run 60 tests (10 tests for each number of processors, plus the baseline test, and the microvisor-only instrumentation). As a result, the workload for each test set was limited to completing in an acceptable amount of time. Instead, we ran with the test data sets.

Publishing results

For each of the official benchmarks run as workloads, we do not show the actual results of the benchmarks. This is for two reasons. The first being that we intended these benchmarks to run as workloads with specific known attributes, not as actual benchmarks of the system. The second is that there are strict guidelines about publishing SPEC results. As we wanted to run these benchmarks on an unsupported platform and build system with various constraints, we acknowledge that we did not comply with these guidelines and therefore cannot publish the results. All SPEC tests were run by taking the source code from a single benchmark and altering it to run in our build system.

6.2.2 Test-set: lmbench

The lmbench workload involved running two copies of lmbench simultaneously.

The lmbench workload generated a massive amount of system calls. So large that, without installing more memory onto the hardware, we could not store full traces for lmbench. As an alternative, we turned logging off for the fourth Linux instance.

Instrumentation effects

The instrumentation overhead for one processor running the lmbench workload is shown in Table 6.8. As expected, they are very marginal. We can see that the microvisor side instrumentation slows the microvisor down by just 0.65%, whilst adding the user side logging only increases that slow-down by a further 0.21%. Given these results are so small, there is no need to look at the instrumentation effects over more processors, as they will become more and more insignificant. This is a testament to the success of our light-weight instrumentation system.

	No instrumentation	**Microvisor only**	**Full instrumentation**
Average	6053499μs	6092545μs	6105645μs
Std. dev.	2.00%	2.01%	2.00%
Overhead	-	0.65%	0.86%

Table 6.8: Instrumentation effects on the lmbench workload.

6.2.3 Test-set: 300.twolf

The 300.twolf workload involved running two copies of 300.twolf simultaneously.

Instrumentation effects

The instrumentation overhead for one processor running the 300.twolf workload is shown in Table 6.9. The effects are even more marginal than those on lmbench, as twolf is a far less system call intensive workload, so it accesses the lock less often. We can see that the microvisor-side instrumentation slows the microvisor down by just 0.35%. The effect of adding the user-side logging is so negligible that it appears to run faster, with only a 0.19% overhead. Given that 300.twolf shows even less instrumentation overhead than lmbench, we can judge that our instrumentation and logging system barely effects non-system-call intensive workloads.

6.2.4 Test-set: 252.eon

The 252.eon workload involved running two copies of 252.eon simultaneously.

	No instrumentation	**Microvisor only**	**Full instrumentation**
Average	8371174μs	8400238 μs	8387131 μs
Std. dev.	1.11%	1.07%	0.77%
Overhead	-	0.35%	0.19%

Table 6.9: Instrumentation effects on 300.twolf workload.

Instrumentation effects

The instrumentation overhead for one processor running the 252.eon workload are insignificant — the results can be seen in Table 6.10.

	No instrumentation	**Microvisor only**	**Full instrumentation**
Average	16779039μs	16750016μs	16790233μs
Std. dev.	0.25%	0.25%	0.29%
Overhead	-	-0.17$	0.07%

Table 6.10: Instrumentation effects on 252.eon workload.

6.2.5 Test-set: eon-lmbench hybrid

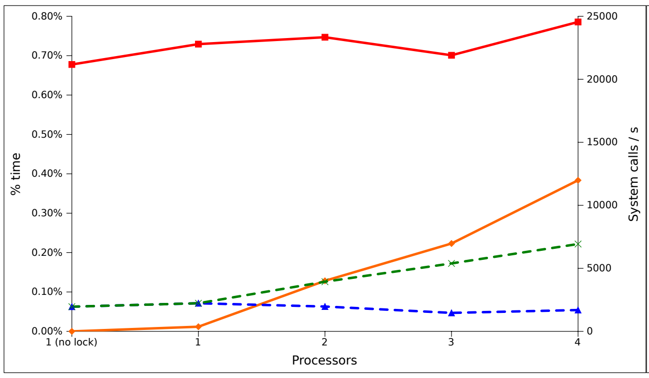
The eon-lmbench workload involved running two copies of 252.eon simultaneously. Since 252.eon has a longer duration than lmbench, we sent a kill signal to the 252.eon process once lmbench finished running.

Instrumentation effects

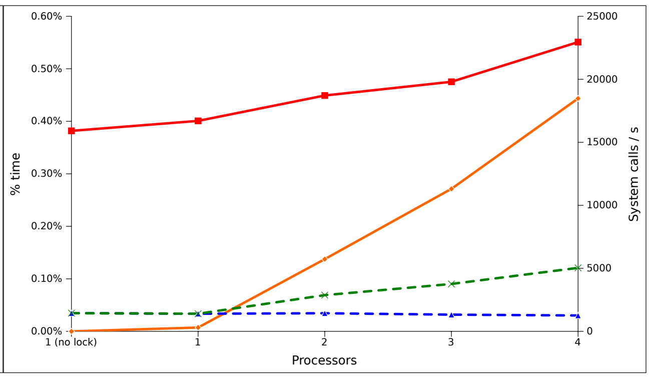
As expected, the eon-lmbench hybrid workload has an instrumentation overhead in between that of 252.eon and lmbench, as can be seen in Table 6.11.

	No instrumentation	**Microvisor only**	**Full instrumentation**
Average	7349366μs	7366735.62305764 μs	7412385μs
Std. dev.	0.96%	1.09%	1.24%
Overhead	-	0.24%	0.86%

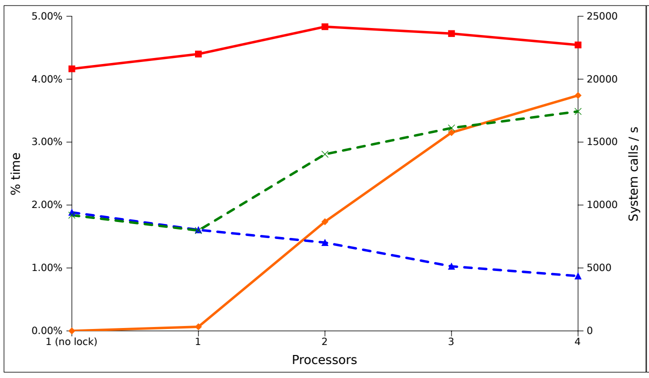
Table 6.11: Instrumentation effects on eon-lmbench hybrid workload.



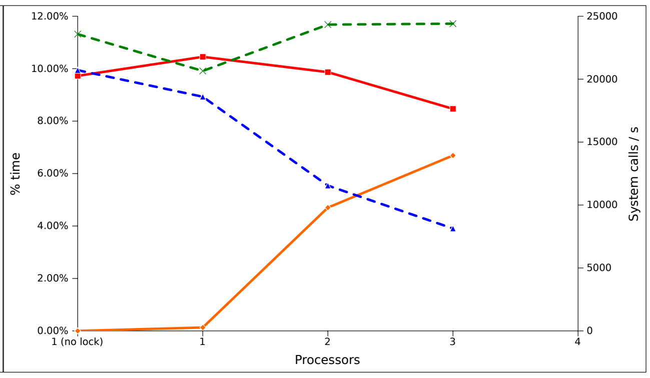
(a) 300.twolf



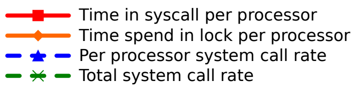
(b) 252.eon



(c) eon-lmbench hybrid



(d) lmbench


Time in syscall per processor
Time spend in lock per processor
Per processor system call rate
Total system call rate

(e) legend

Figure 6.3: Characteristics of each workload, over 1 to 4 processors. Note that the left y-axis of each graph does not use the same scale; since the larger workload ranges would dwarf the smaller workloads.

6.2.6 Workload analysis

The following section documents the general features and compares the different workloads over 1 to 4 processors. Note that unless otherwise noted, all figures are averaged from four independent trace files.

Figure 6.3 shows how the different workloads compare to each other in general. Note that the zeroeth point on the *x*-axis shows the workload running on one processor without the lock, and the following point shows the workload running on one processor with the lock. This is to show the impact of lock latency on workload duration.

Each chart shows the change in the following:

- Per-processor system-call rate.
- Total system call rate.
- Time spent in lock per-processor. This refers to the amount of time spent obtaining the lock or waiting on the lock.
- Time spent in system calls per processor.

Time in the lock increases consistently as we add more processors. System call rate per processor stays flat, as long as the total system call rate does not decline. The CPU-intensive workloads barely show any effect of having no lock and an uncontended lock, whilst the system call intensive workloads, being already at their maximum system call rate, suffer degradation.

Contention

To undertake further analysis, we turn to looking at how contention increases as we add more processors. Contention as a measure here is not how contended the lock was, as we cannot sample this all the time. Rather, contention refers to how often the lock was taken when a processor first attempted to take it.

Figure 6.4 shows the average contention values for each workload. System call intensive workloads suffer the most, with lmbench contention reaching 80%.

Contention rates seem very high in comparison to the amount of time each processor spends executing system calls or in the lock. Consider the fact that the 300.twolf workload reaches nearly 20% contention on two processors, when less than 0.5% of each second is spent in the lock, and less than 0.7% of each second is spent executing system calls. The high contention is due to two things:

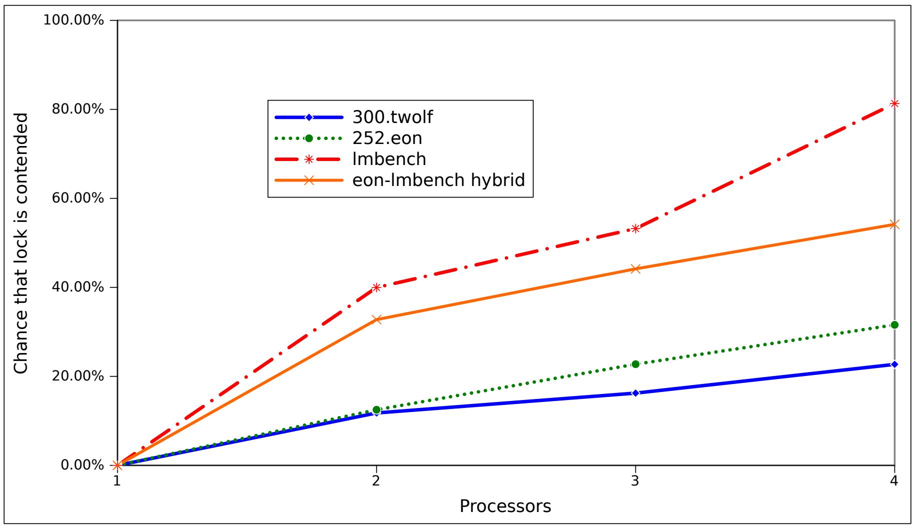


Figure 6.4: This graph shows, for each workload over 1–4 processors, how likely the lock was to be contended when a processor tried to take the lock.

- Each Linux instance performs context switches on a timer tick from the timer server. Each Linux receives this tick at the same time. Therefore every Linux instance attempts to execute all of the microvisor system calls to perform the context switch at approximately the same time.
- The workloads are being performed simultaneously, meaning that any system calls a workload makes will happen at approximately the same time.

A consequence of these two features of the workloads is that we could expect less contention from a real workload with the same system call rates, as a real workload will not execute all system calls at virtually the same time. Less contention implies less time wasted waiting on the lock, and, by extension, we can expect greater scalability from workloads with the same system call rates.

System call rate

Although the system call rates are shown in Figure 6.3, we present them again in Figure 6.5 to highlight the differences between the workloads. Both of the SPEC based workloads exhibit fairly similar system call rates, whilst lmbench executes nearly 24,000 microvisor system calls per second. Recall that Linux system calls, although they do enter the microvisor, do not take the global lock and are not locked, so the rate we see here is purely system calls triggered by context switching and Linux making system calls to the microvisor.

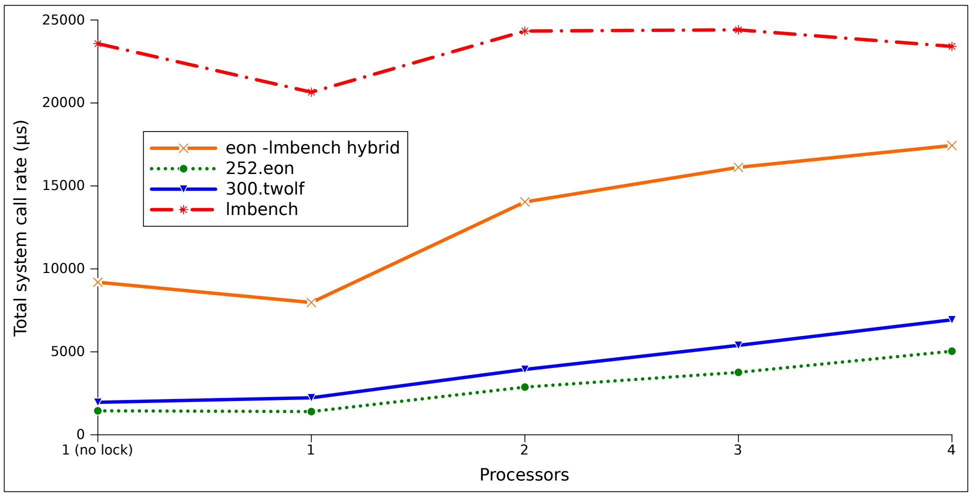


Figure 6.5: This graph shows how the total system call rate varied between workloads.

The hybrid workload shows a system call rate approximately half-way between the lmbench workload and the SPEC workloads. It is interesting to note that adding the lock decreases the system call rate of the hybrid workload, even though the system call rate is not nearly as high as lmbench. This could be because the hybrid load performs more expensive system calls.

Scalability

Probability of contention and the system call rate directly affect scalability. Figure 6.6 shows a normalised scalability curve for each workload. To achieve a measure of throughput, we normalised the time taken for workloads over 2 – 4 processors by the time taken to execute the uniprocessor workload. The graph shows that the hybrid workload scales acceptably up to two processors, with two processors executing roughly one and a half workloads in the time taken for one processor to execute the entire workload. Both of the SPEC workloads exhibit nearly ideal scalability, whilst the hybrid workload scales effectively to 2 processors, but degrades sharply at three processors.

Figure 6.7 shows the relation between system call rate and throughput, by taking the line of best fit for the throughput of each workload vs. total system call rate per second. This chart allows us to estimate the scalability of a workload based on the number of processors and the system call rate. Of course this will not allow us to pinpoint exactly how scalable a workload will be, as a workload that makes many expensive system calls will scale less than a workload with the same system call rate but faster system calls.

An important result here is that Figure 6.7 shows what is the worst case for scalability of a work-

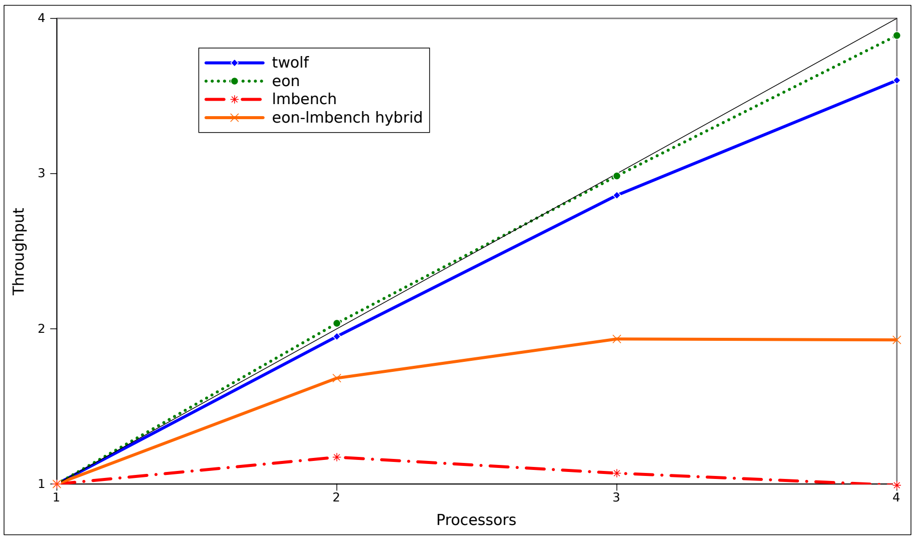


Figure 6.6: This graph shows a normalised scalability curve for each workload. It is based on the time taken to complete the workload. A reference line is shown for ideal scalability.

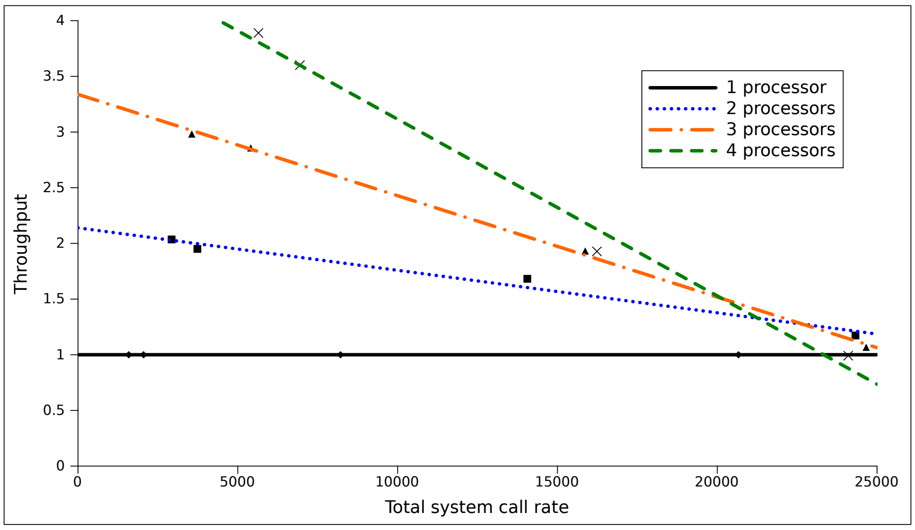


Figure 6.7: This graph shows throughput vs. total system call rate and illustrates the relationship therein.

load at that total system call rate. As mentioned previously, due to the way we ran our workloads, all system calls happened at the same time, sharply driving contention values up. What this chart allows us to take the a system call rate and an amount of processors (between one and four) and predict how that workload will scale in the worst case.

6.3 System call analysis

As the system calls themselves form the critical section for the giant-locking implementation, we undertook an investigation into how fast the system calls are and how often they are executed. We used the trace files from lmbench test-sets to conduct the system call analysis, as lmbench is the most system call intensive workload and would give us insight into the worst-case scenario.

When analysing system calls, it is important to note that Linux system calls do not acquire the giant lock. Whilst they do enter the microvisor, the microvisor redirects Linux system calls back to the guest operating system before lock acquisition.

System call number	**System call**
A	Cache flush range
B	Channel primary free message
C	Channel primary receive
D	Channel secondary send
E	Interrupt ack
F	Interrupt register
G	Interrupt unregister
H	Interrupt wait
I	Interrupt wake up
J	VFP get FPEXC
K	VFP set FPINST
L	vIRQ raise
M	VMMU activate VUAS
N	VMMU flush range
O	VMMU map segment
P	VMMU translation insert
Q	VMMU translation invalidate

Table 6.12: System call number and definition

The set of system calls used by the lmbench workload (and all of the other workloads) is shown in Table 6.12 for quick reference.

6.3.1 System call duration

Figure 6.8a shows the average length of each system call for a one processor lmbench workload. By looking at this table we can see that most system calls are limited to a few microseconds or less on average.

Figure 6.8b shows how system call duration increases as we add more processors. System call duration can increase for several reasons; first, more processors could make microvisor data structures and algorithms in system calls take longer to execute, due to increased complexity. We do not attribute this reason to much of the increase in system call duration as the microvisor system calls are designed to be fast and efficiently.

Another more serious reason for system call durations to increase is cache coherency between the processors. As we increase the amount of processors, there is a higher probability that the processor executing a system call will not be the same processor that previously entered the microvisor. Additionally, as the microvisor has a very small amount of shared resources, nearly every system call touches similar resources, so the likelihood of the same resources being used by different system calls is much higher.

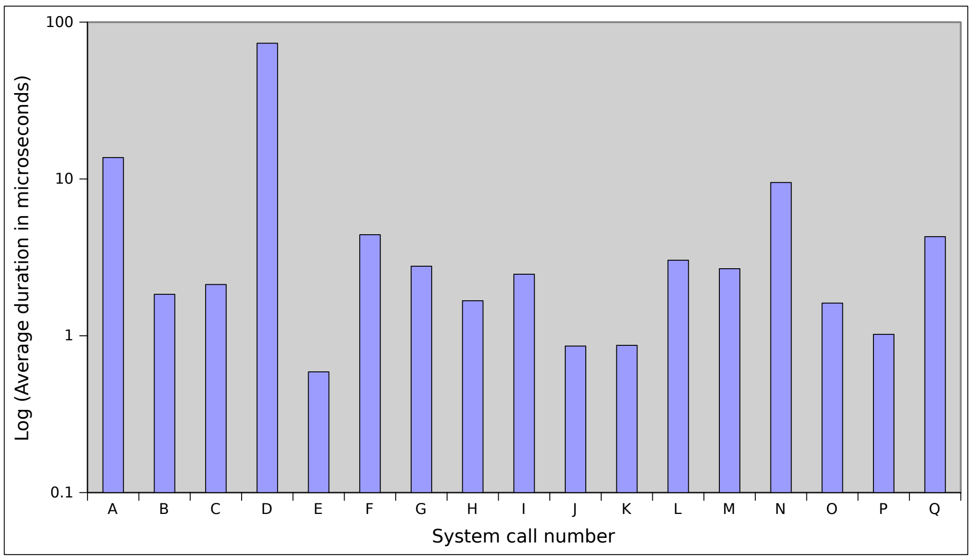
One system call, (D — channel secondary send) actually gets shorter as we add more processors. Presumably this is because that system call does not touch many shared resources, or is frequently executed by the same processor repeatedly.

Note that running eight instances of lmbench over four Linux instances is a massive stress test for the scalability of the microvisor. All lmbench instances are repeating sets of the same system call over and over again, so by four processors we are probably alternating between executing the same system call on different CPUs nearly every single time we enter the microvisor. This means that cache contention is as high as – if not higher – than lock contention.

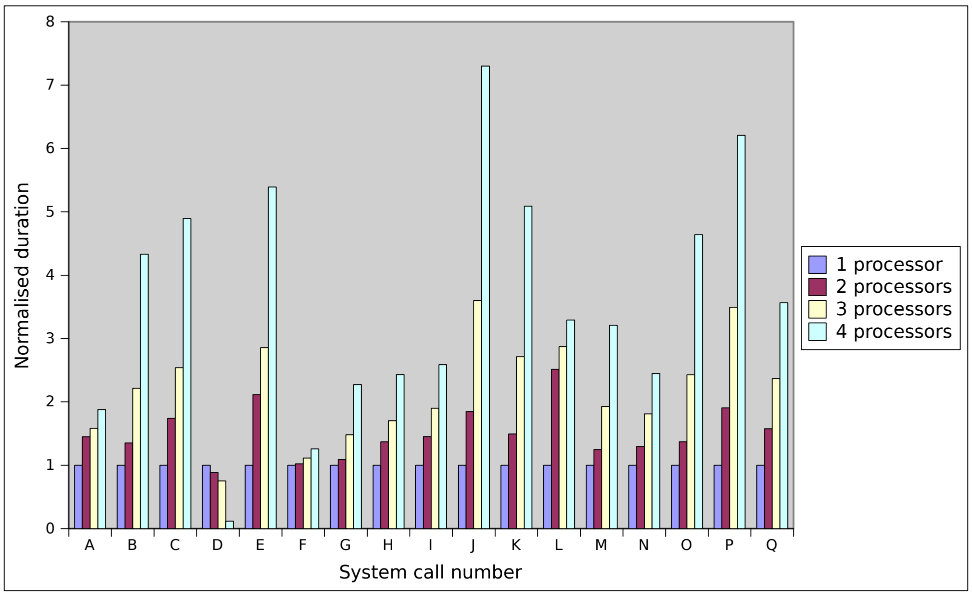
The fact that system calls get significantly longer in duration has the following implications. Firstly, it means that the poor scalability of lmbench is not only due to the giant lock, but also due to very high cache contention. Secondly, it indicates that adding a finer granularity of locking would probably not improve scalability in this case. Adding a finer granularity of locking would increase the system call duration and number of resources touched by a system call. All CPUs attempting to execute the same system call simulateneously would then need to migrate more cache lines to execute the system call, which would negate the potential improvements offered by a finer granularity of locking. This would only be solved by avoiding shared resources altogether.

6.3.2 System call occurrences

System call average duration lacks significance as a figure without knowing how often those system calls are made. Figure 6.9 shows, on average, how many system calls are made during one lmbench workload. From this we can see that the system calls are far from equal when it comes to occurrences. Cache flushing accounts for a massive amount of the system calls made, with vMMU operations following closely behind.

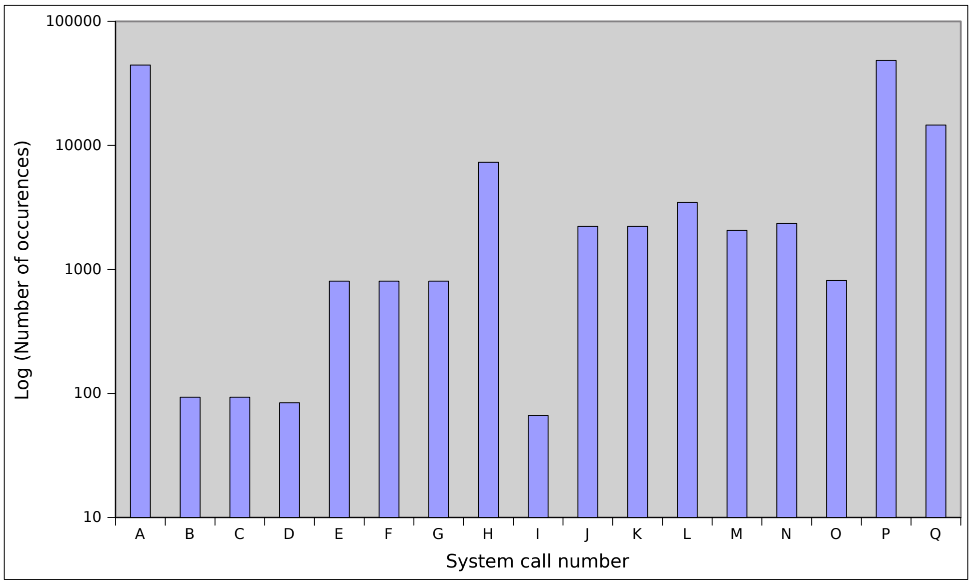


(a) Average system call durations with one processor

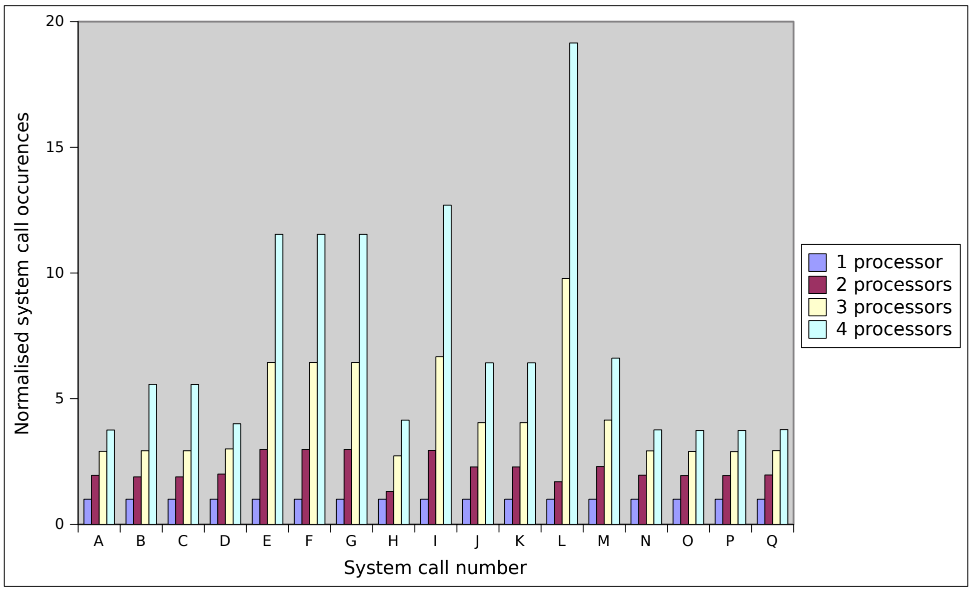


(b) Normalised average system call durations for 1–4 processors

Figure 6.8: These two charts show the average durations of system calls made during an lmbench workload and how the average system call durations increase as we increase the amount of workloads and processors. **Note the values for four processor lmbench are missing system call counts for one Linux instance**



(a) Lmbench system call occurrences with one processor.



(b) Normalised system call occurrences for 1–4 processors.

Figure 6.9: These two charts show the amount of system calls made during an lmbench workload execution and how these values increase as we add processors and increase the amount of workloads. **Note the values for four processor lmbench are missing system call counts for one Linux instance**.

6.3.3 System call impact

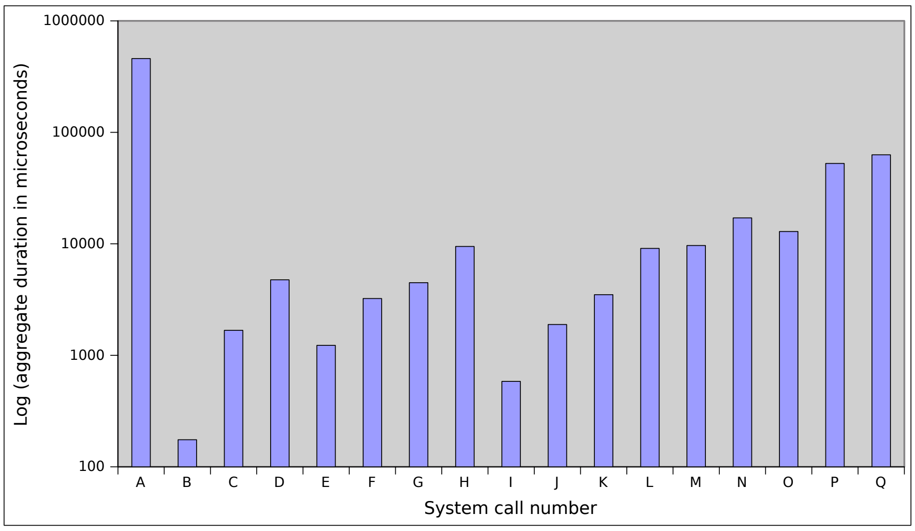


Figure 6.10: This chart shows the total amount of time spent in each system call during a one processor lmbench workload.

System call frequency and duration are not very significant values when taken alone. Whilst it is preferable to have the fastest system calls possible, a long running system call that occurs very rarely will not impact the system as greatly as a medium length system call that occurs very frequently. In order to isolate the system calls that impact on scalability the most, we looked at the total time each system call takes in the system over a one-processor lmbench workload. The results are shown in Figure 6.10.

The results of this analysis show us that cache flushing (system call A) accounts for the majority of the time spent in the kernel critical section, followed by interrupt related operations (system calls P and Q). The system call that is longest on average (D) does not significantly impact the aggregate time spent in system calls at all.

These results are significant as the most expensive system calls — VMMU and cache flushing — are vCPU specific and do not necessarily need to be locked. Neither category of operation involves modifying another thread's state.

A conclusion that can be made here is that we could definitely improve scalability significantly by removing expensive and/or frequent system calls that do not touch shared state from the lock. This would be a minor change for large scalability increases.

Further detail about why some of these system calls appear to scale poorly on this platform with this version of OKL4 are presented in Section B.1.

6.4 Simulation

6.4.1 Simulator accuracy

Before running any simulations, we first needed to establish and test the accuracy of the simulator on reproducing real workload runs. To do this, we ran the simulator on *n* processors, on trace files for *n* processors for that workload and compared the time taken according to the simulator with results for running on real hardware. We ran the test with mode switch duration and lock entry latency set to the figures we calculated earlier in Section 6.1.

Processors	**Actual duration (μs)**	**Simulated duration (μs)**	**Difference**
		300.twolf	
1	8449333	8472766	0.28%
2	8663408	8682286	0.22%
3	8862311	8869829	0.08%
4	9386267	9472870	0.92%
		252.eon	
1	16788071	16797144	-0.05%
2	16495589	16509261	-0.08%
3	16873128	16841550	0.19%
4	17263644	17221493	0.24%
		eon-lmbench hybrid	
1	7901608	7586949	4.15%
2	9394662	8880326	5.79%
3	12254322	11937175	2.66%
4	16389768	15973662	2.60%
		lmbench	
1	6104772	6181006	-1.23%
2	10410112	10118153	2.89%
3	17122117	16289354	5.11%

Table 6.13: Accuracy of the simulator at replaying traces.

Table 6.13 shows the results. We see that the simulator is very accurate for low system call loads, but loses accuracy as we use longer loads with higher system call rates. It is also more accurate for loads with less processors. The main reason for this is that lock latency is taken as a constant, and does not change according to cache effects. Other reasons for simulator inaccuracy are documented in Section 5.2.2.

Note that for the lmbench workload we do not test four processor accuracy. This is due to the fact that we could not store whole logs for lmbench on four processors, due to memory limitations, so any simulation based on those logs would be essentially inaccurate.

What these results show is that for a simulator that simply replays events and simulates locks, without showing any cache effects, for the workloads that we play we achieve reasonable accuracy.

6.4.2 Simulated scalability

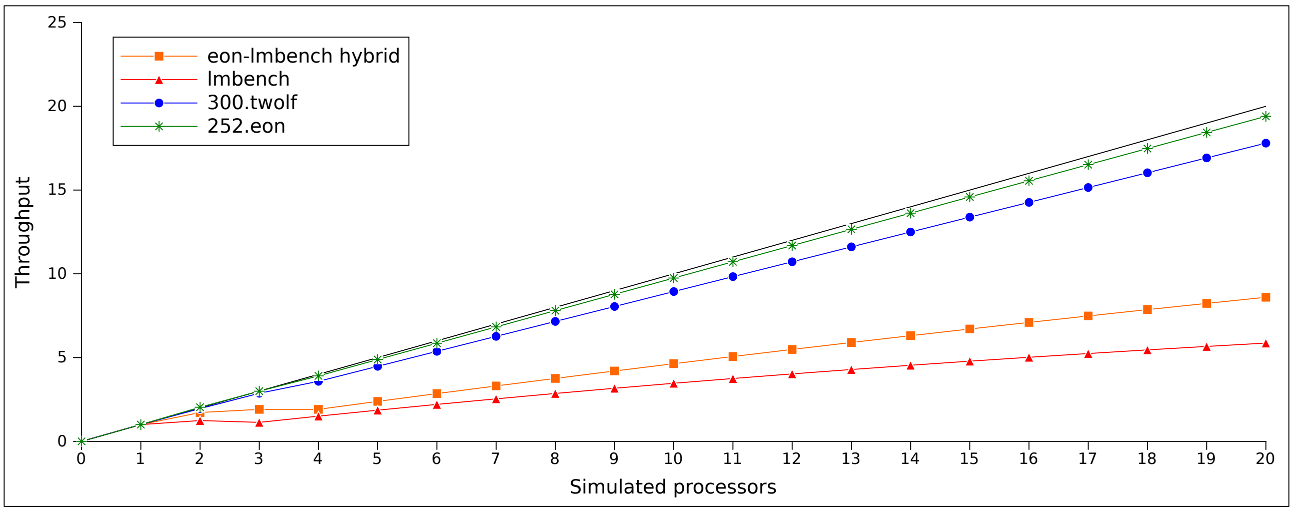


Figure 6.11: The simulated scalability of all four of the workloads.

We used the simulator to attempt to see how each of the workloads would continue to scale up on more and more processors. The first thing we notice is that the simulator indicates that the workloads would scale much further than we expect based on the actual results for one to four processors. To test how accurately the simulator could scale, we used the simulator to scale a three processor workload up to a four processor workload and compared the % increase in time taken to execute the workload. Table 6.14 shows the results, from which we can conclude that whilst the simulator does show an increase in time taken, it is an order of magnitude out when it comes to how much slower a workload should complete.

The results from the simulator with the implementation as described in Section 5.2 were actually worse at scaling than the results described above. In light of the altering system call durations, we altered the simulator to scale system call duration by the difference between the average three-processor and four-processor system call durations, for each individual system call for each workload. This is not realistic as we have no model for how system call duration would change as we add more and more cores. The increase between three-processor and four-processor system call durations is not ac-

Workload	Actual % increase	Simulated % increase
300.twolf	5.91%	0.38%
252.eon	2.31%	0.25%
eon-lmbench hybrid	33.75%	3.11%
lmbench	43.78%	6.89%

Table 6.14: Inaccuracy of the simulator at scaling workloads.

curate. Presumably, it would increase until we had reached a point of saturation where all system calls are executed by alternating processors. At this point system call duration would probably stabilise.

In any case, the simulated results are shown in Figure 6.11. The fact that the simulator is far too optimistic is illustrated in the bends in the curves. One can see this in the plots for the lmbench workload and the eon-lmbench hybrid workload, as they both reverse direction at the third and fourth processor, respectively.

The simulators failure at accurately scaling workloads provides us with an unexpected result. Based on the previous results shown on system call duration increase, we see that cache contention dominates the scalability of our workloads far more than the lock implementation does. If the giant-locking scheme were the only thing effecting scalability our workloads would be much closer to the ideal scalability curve.

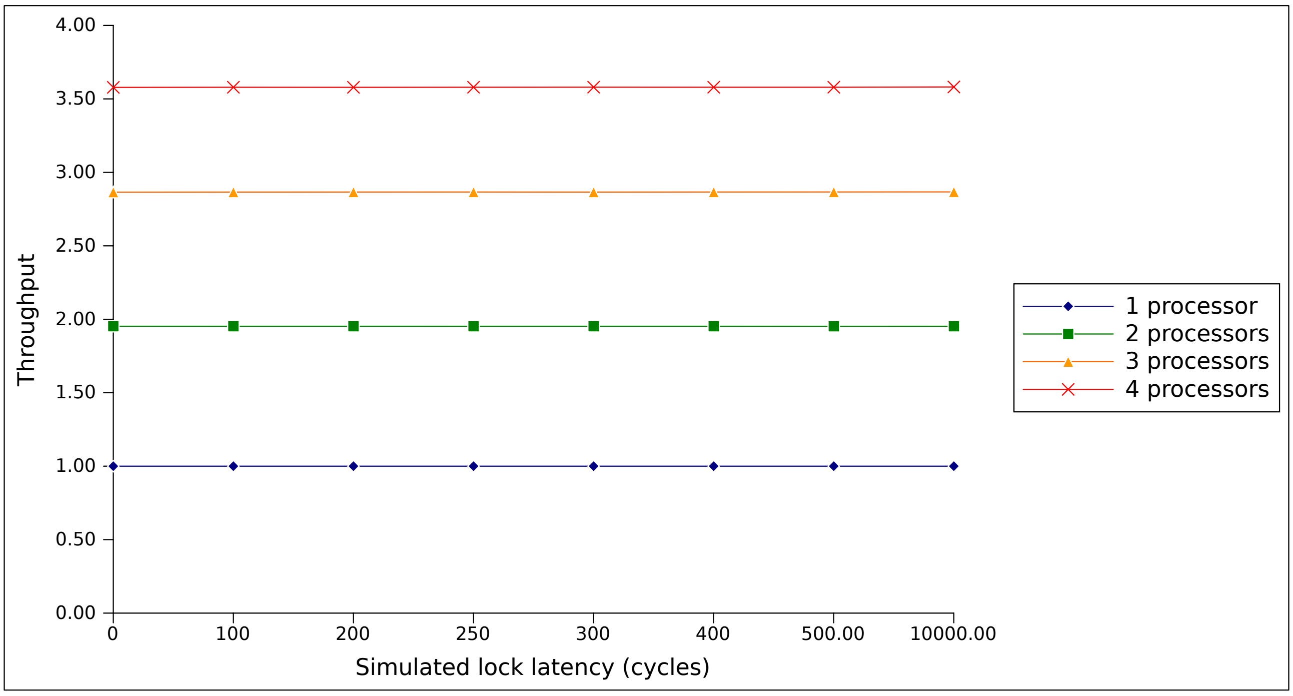
6.4.3 Simulated lock latency

The next experiment conducted with the simulator had the purpose of investigating the effect of lock latency on scalability and workload duration with a giant-locking scheme. We used the existing traces for each workload and specified a custom lock latency to the simulator. Recall that when the simulator is given a custom lock latency, it splits the latency into $\frac{2}{3}$ lock entry and $\frac{1}{3}$ lock exit.

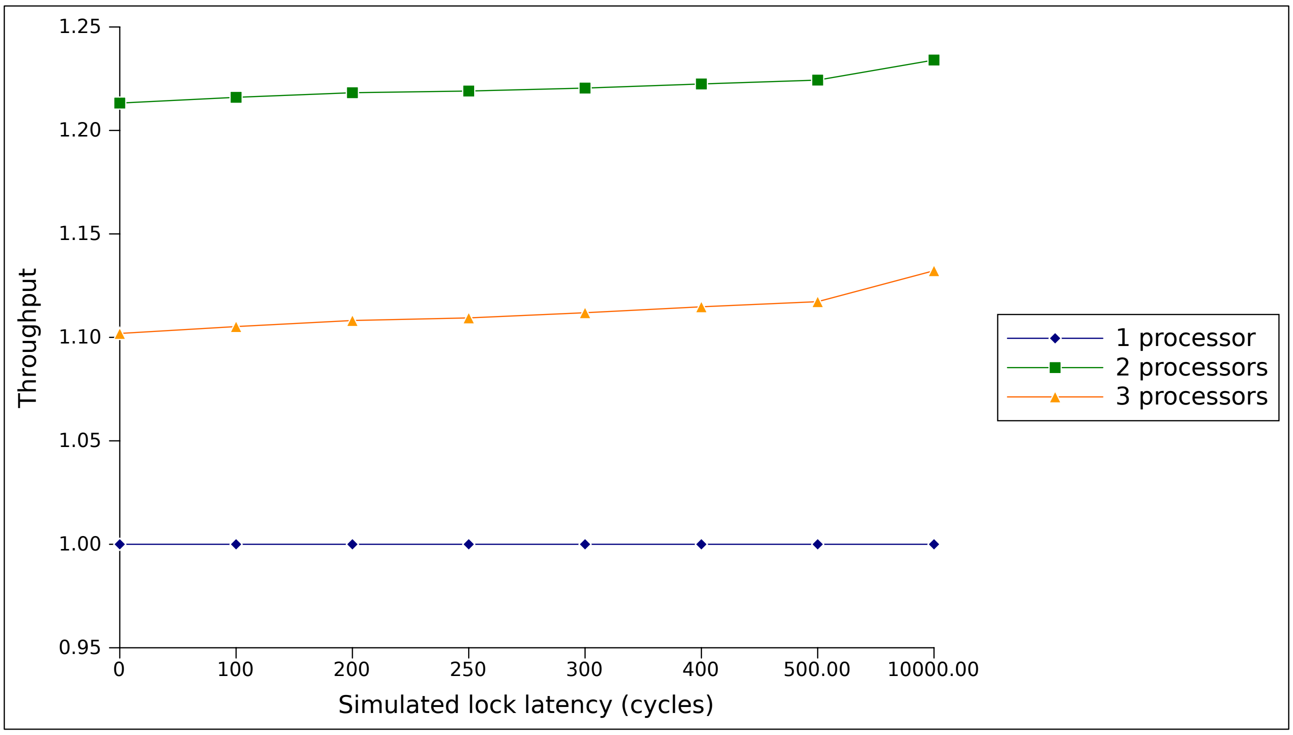
Figure 6.12 shows the effect of lock latency on scalability of a workload. We normalise each duration and show how the workload would scale given the specified lock latency. For a CPU-intensive workload (300.twolf) lock latency has no effect on how the workload scales, as even 1000 cycles shows no change. 1000 cycles is about as long as many of the average system call durations, so the lock entry and exit would approximate to being as long as the critical sections.

For a system call intensive workload, we see a minor impact as we increase the lock latency.

Just because a workload scales well does not mean that it executes efficiently. Figure 6.13 shows the effect of lock latency on workload duration. We can see that as we add more processors, lock latency becomes more irrelevant, as the workload is affected more by contention than locking costs.

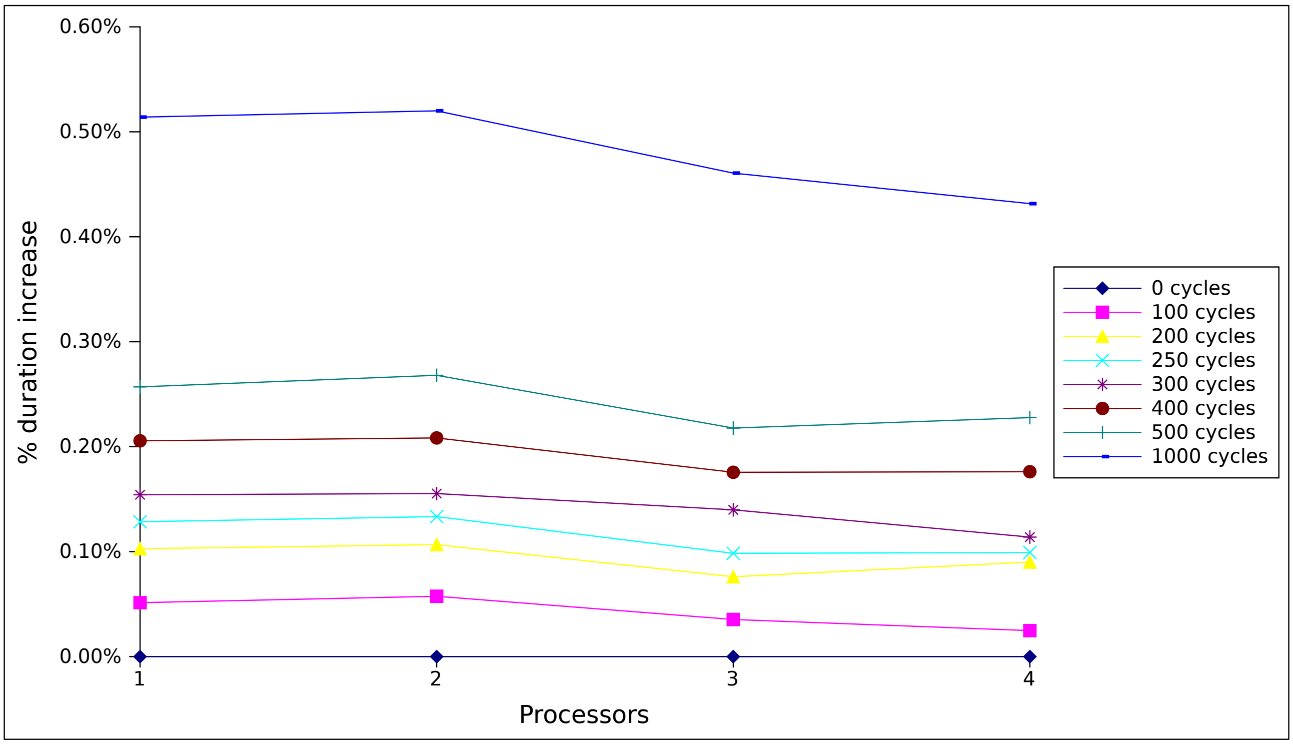


(a) 300.twolf

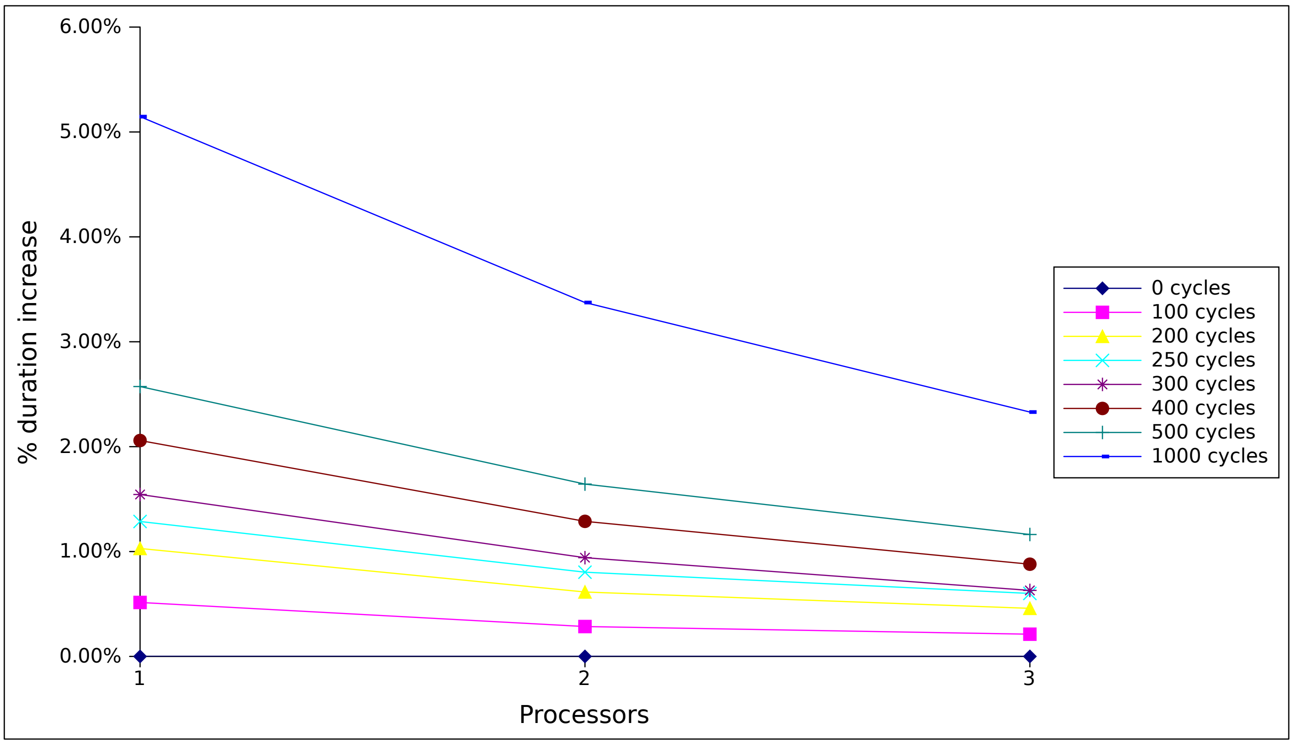


(b) lmbench

Figure 6.12: These two charts show the effect of a changing lock latency on the scalability of a workload.



(a) 300.twolf



(b) lmbench

Figure 6.13: These two charts show the effect of a changing lock latency on workload duration.

This result shows that the 250 cycle overhead of the OKL4 microvisor lock implementation is hardly significant when it comes to how workloads scale.

6.4.4 Simulated coarse grained locking

Based on the result that system call lengths greatly increase as we add more processors, the next step was to simulate a basic coarse-grained locking scheme. First we established a practical model of what coarse-grained locking could look like for the microvisor, however it is only intended to be a sample. We reasoned that the only system calls that essentially need to be locked are those that modify other thread resources. These system calls include vIRQ raise, channel operations and scheduler operations.

We then reasoned that scheduler data structures would require one lock, as all processors could attempt to access it. For system calls that modify another threads resources, we would need a lock per thread control block.

Based on this model, only four system calls out of the set of system calls made by the Linux instance would require locking. The first three are channel operations and the last is vIRQ raise.

Channel system calls

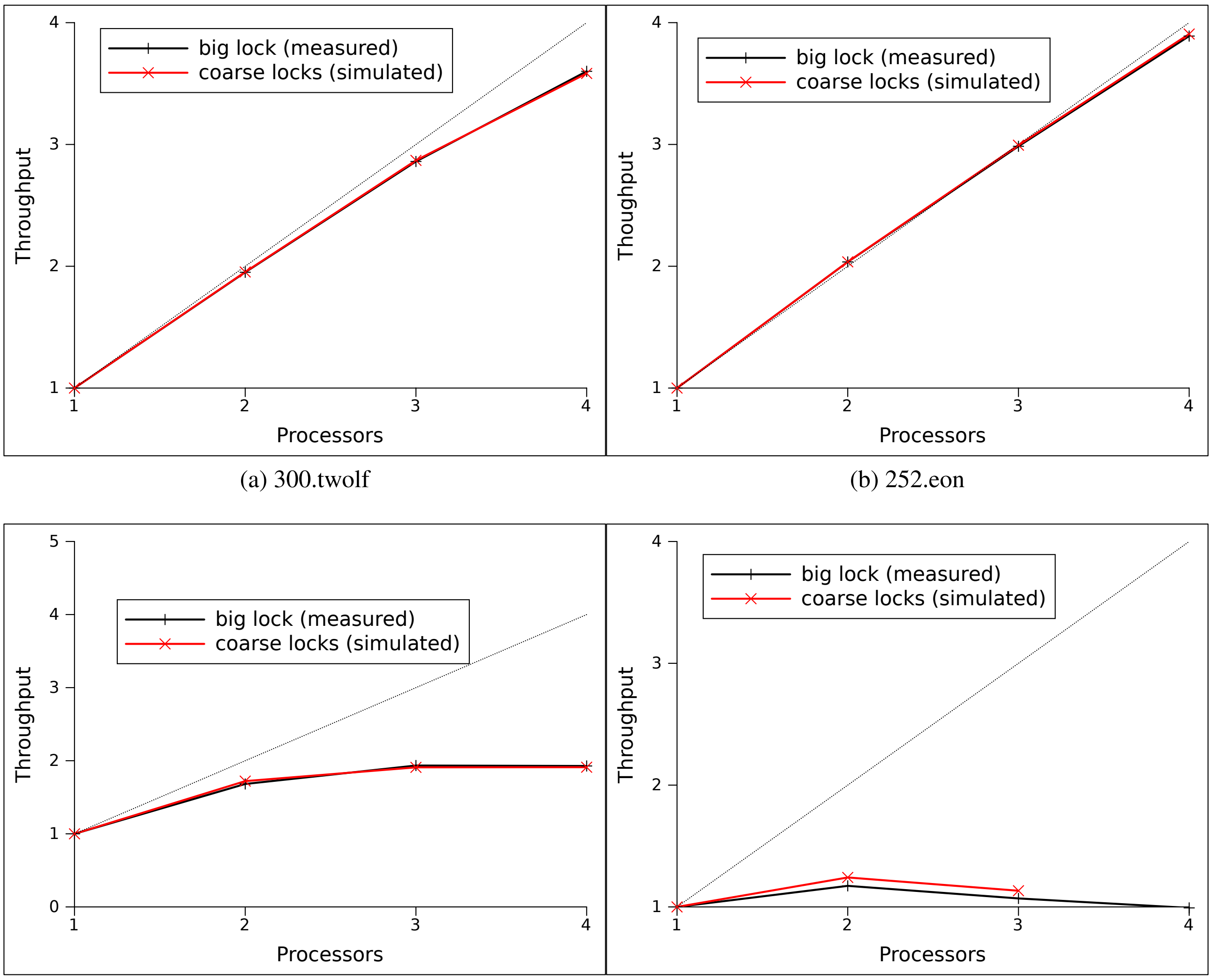
Recall from Section 5.2 that for each row in the log, we generate a list of events for the simulator to simulate. In order to generate an event queue from a log row for a channel system call, we did split the system call into three segments as follows:

- The first segment is the part where we modify another the data of another thread. So we insert a lock event. As this is a per-thread lock, we need to choose a thread. We reasoned that channel operations are only happening between Linux instances and the serial server, so if the application ID (taken from the log) making the system call is the serial server, we choose a random Linux ID to lock. Otherwise, we choose the serial server lock. We push a system call event for 40% of the system call duration onto the event queue, and an unlock event.

- The second segment is lock free, so we simply push a system call event onto the event queue, for 20% duration taken from the log.

- The third segment is the part where we access the global scheduler queue. We add a lock event for the global scheduler lock, a system call event for the remaining part of the system call and an unlock event.

The values for splitting up the system call are fairly arbitrary, as we have not conducted any analysis into how the system calls are split up.



(a) 300.twolf

(b) 252.eon

(c) eon-lmbench hybrid

(d) lmbench

Figure 6.14: The modelled scalability of the workloads using coarse grained locking.

vIRQ raise system call

The vIRQ raise system call is implemented in the same way, with the same proportions as the channel system calls with one difference: the selection of the per thread lock. We reason that vIRQ raise is being sent to the Linuxes by the timer server, so we pick a random Linux ID to lock.

The simulator used the same lock latencies as calculated in the micro-benchmarks.

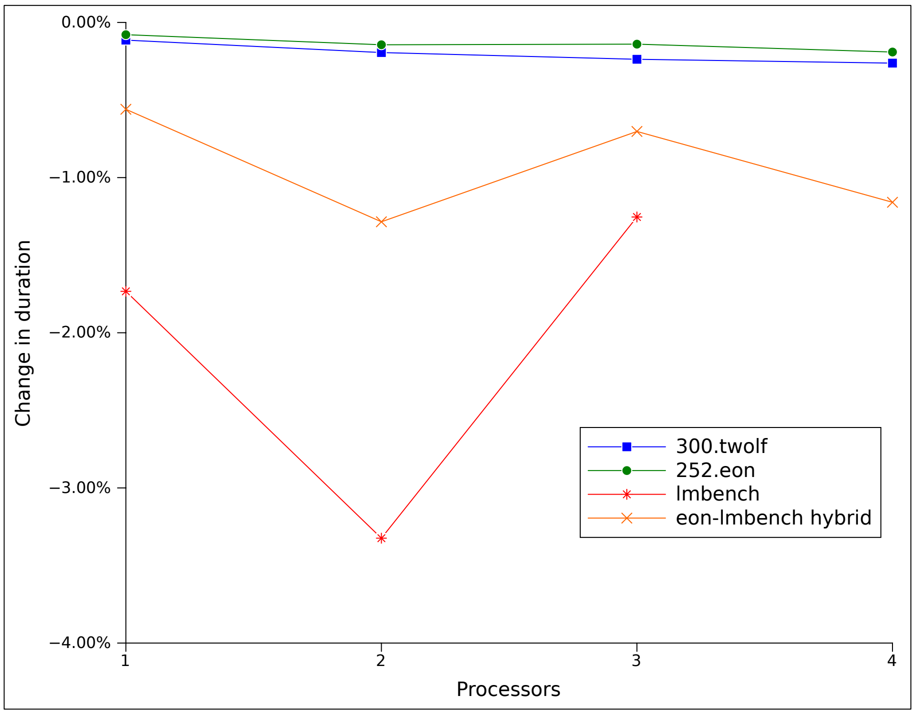


Figure 6.15: The modelled effect of coarse grained locking on workload duration. In all cases it reduces the workload time, but not by much.

Simulation

We used the coarse-grained locking model to simulate two things: how such a model would change the scalability and duration of our workloads over 1 – 4 processors and how such a model would change how the simulator scaled workloads.

Figure 6.14 shows the change in scalability for each workload. The difference between these simulations, and the simulations used to show simulator accuracy is only that the locking scheme is different. Apart from lmbench, we see no significant change in how scalable the workloads are. As we would expect that making the most expensive system calls parallellisable (i.e. not locked), this reinforces our previous result, and shows that cache contention is halting scalability far more than the giant lock.

A change can be noted in Figure 6.15, which shows how the duration of the workloads changed when simulated under the coarse-grained locking scheme. The coarse-grained locking approach appears to marginally reduce the duration of the workloads, especially for the system call intensive workloads whilst they continue to scale.

Figure 6.16 shows the result of simulating the coarse-grained locking model on more processors.

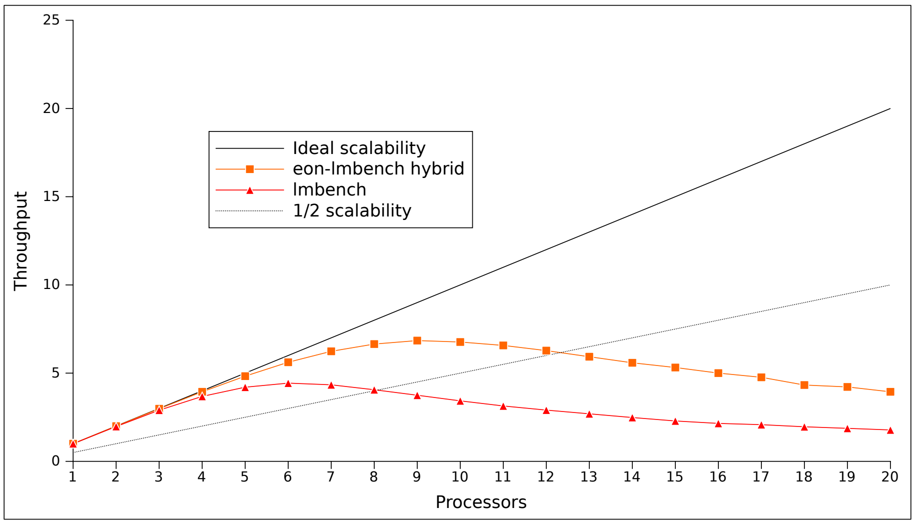


Figure 6.16: The simulated scalability of all four of the workloads with a simulated coarse grained locking scheme.

Contrary to the minimal effects coarse-grained locking had on simulating the 1 – 4 core loads, the impact increases as we add more processes, and indicates that coarse grained locking would be worth further investigation if higher scalability were desired. We can conclude that this is because as we add a large amount of processors, lock contention becomes the dominating factor in scalability over cache contention. This would be even more true in real experiments, as once we reach a saturation of system calls all executing on alternate processors, they would cease to increase in duration.

Limitations

The coarse-grained locking simulation is intended as a basic model only and has many limitations. Several of them are enumerated below:

- The multiple locks in the coarse-grained locking simulation behave the same as the locks are modelled according to the giant-lock, which would not necessarily be implemented in the same way.
- The implementation seems practical, but would probably be a lot more complicated in practice.
- The coarse-grained locking simulation is limited by the aforementioned inaccuracies of the simulator.
- The coarse-grained model does not include any increase in system call duration that may occur as a result of introducing more locking.

6.5 Real workloads

Having determined a relationship between scalability and system call rate for the giant-locking scheme, the next step was to relate this to real workloads. We implemented a mechanism in the microvisor to count Linux system calls by incrementing a memory address in the log buffer region. This was possible as the microvisor redirects system calls back to guests just before entering the locking code.

Workload	**Linux system calls**	**Microvisor system calls**	Average duration (s)
252.eon	343	14261	16.7
300.twolf	755	17607	8.5
lmbench	17450	125152	6.1
eon-lmbench hybrid	9594	67100	7.9

Table 6.15: Total number of Linux and microvisor system calls for each workload, and their duration in seconds.

Given the average total number of Linux system calls made for each workload, the average total number of microvisor system calls made for each workload and the time taken for each workload, we were able to define an approximate model for estimating the number of microvisor system calls made per Linux system call. We reasoned that there is a base rate of microvisor system calls made by Linux when context switching as well as more microvisor system calls made per Linux system call. The relation is expressed as follows:

$$\text{TotalMicrovisorSystemCalls} = \alpha \times \text{WorkloadDuration} + \delta \times \text{LinuxSystemCalls} \tag{6.1}$$

Of course this is for estimation only, as different Linux system calls will make different amounts of system calls. We solved this equation simultaneously using the values in Table 6.15 to find that α was approximately 0.001 for workload durations expressed in microseconds. We got an average value of 7 for δ.

The second stage in this analysis was to find the Linux system call rates for some real workloads on an embedded device. We took an Android phone running the Android debugger and native Linux function tracing to calculate the number of Linux system calls made over some workloads. The specific workloads used were:

- An mp3 playing.
- Recording video.

- The phone idling with the screen on.

We counted the number of Linux system calls made, first stripping out any system calls made by the debugger. The results are shown in Table 6.16. Based on our rough estimate, the Android phone made approximately 67 Linux system calls per second whilst idling and up to 8614 system calls per second whilst recording video. According to our approximation, this corresponds to a minimum of 1480 microvisor system calls per second, up to 13059 per second.

Workload	Linux system calls/s	Predicted microvisor system calls/s
idle	67	1480
mp3	1117	8819
recording video	1723	13059

Table 6.16: Linux system call rates for the Android workloads and predicted microvisor system call rates.

Looking back at Figure 6.7 which shows the correlation between system call rate and approximate scalability, the idle system call rate correlates with the system scalability of the 252.eon workload. As such we could expect the idle Android workload to scale well using giant-locking for up to and over four processors based on our results.

The system call rate for video playback is just over that of the eon-lmbench hybrid workload, meaning that, were our estimation accurate, video playback on multiple processors at once would not scale well beyond two processors.

However, there are several reasons why we could expect system call loads such as video playback to scale perfectly well should they run on a multiprocessor platform using the OKL4 microvisor:

- Our relation between system call rate and throughput assumes that the total system call rate is shared equally between all processors. In a real workload, it is likely that only the processor playing video would be making the bulk of the total system call rate. A processor cannot contend with itself, so better scalability would be seen.

- Our relation between system call rate and throughput is the worst possible case for cache contention, as we execute the same system calls at the same time, meaning processors constantly performing expensive cache line refills and other cache coherency protocols.

- It is unlikely that one would perform several system call intensive workloads — such as playing music and video — at the same time.

From these results we can confirm our initial hypothesis. That the giant-locking scheme is suitable a real, multiprocessor embedded device.

Chapter 7

Conclusion

In undertaking this thesis, we have developed a light-weight instrumentation system that provides a wealth of data at incredibly low overhead. We have found that the current OKL4 microvisor lock implementation adds a 420% overhead to the normal system entry & exit cost, but that according to simulation this cost does not significantly impact scalability. We have established that, on our experimentation platform, cache contention has a much higher impact on scalability than locking scheme.

We showed that one very expensive system call accounts for the majority of the time spent in the microvisor, and that this system call could be taken out of the locked code, in addition to other, shorter and less frequent system calls.

Additionally, we implemented a discrete event simulator that could accurately replay system call traces and be used to investigate further scalability. We found that, according to simulation, eventually coarse-grained locking can offer higher scalability, but only once cache contention had been overtaken as the dominating overhead by lock contention.

Finally and most importantly, we show that our initial hypothesis is true. That on a high performance microvisor running on high performance hardware, giant-locking is a simple approach that scales perfectly well for one to four processors. Through simulation, we showed that it can probably scale a lot further, although attributes of the hardware would need to be known to make this claim more concrete.

7.1 Future Work

Our results are specific to one hardware platform and one microvisor. A large project for future work would be to undertake a more generalised approach to make broader claims. Testing on a wide range of embedded hardware with different devices would allow for more complicated workloads using a wide range of devices.

Another avenue for future work would be to undertake an implementation of the coarse-grained locking scheme outlined in Section 6.4.4 and run the same benchmarks. It would also be ideal to design some workloads that do not execute system calls at exactly the same time to establish at what scalability is dominated by cache contention.

Further, a more accurate model for a simulator could be built that modelled cache contention accurately so that we could determine more realistic results for scalability on greater than four processors.

Appendices

Appendix A

Instrumentation effects

Lock instrumentation effects on a highly contended lock over 2 – 4 processors. Cycle measurements are taken every 64th cycle. Standard deviations are shown relative to the averages.

Processor ID	1		2		3		4	
Column	Cycles	Instr.	Cycles	Instr.	Cycles	Instr.	Cycles	Instr.
2 processors without instrumentation								
Average	6617003	1163311	7060730	1214266	-	-	-	-
Std. dev.	0.33%	1.23%	0.86%	1.43%	-	-	-	-
2 processors with instrumentation								
Average	6493004	1375947	6920749	1482438	-	-	-	-
Std. dev.	1.79%	5.63%	1.88%	6.21%	-	-	-	-
Overhead for 2 processors								
Overhead	-1.87%	18.28%	-1.98%	22.09%	-	-	-	-
3 processors without instrumentation								
Average	9677216	2048144	10257258	2320656	9965083	2120338	-	-
Std. dev.	2.22%	1.83%	2.45%	1.78%	2.22%	1.89%	-	-
3 processors with instrumentation								
Average	10413419	2293553	10757474	2259934	10982411	428766	-	-
Std. dev.	1.60%	1.21%	1.53%	2.06%	2.90%	3.87%	-	-
Overhead for 3 processors								
Overhead	7.61%	11.98%	4.88%	-2.62%	10.21%	14.55%	-	-

4 processors without instrumentation								
Average	13078517	3376107	13670995	3512628	13399679	3449006	13464855	3471763
Std. dev.	1.73%	3.15%	1.87%	3.34%	1.80%	2.97%	1.72%	2.75%
4 processors with instrumentation								
Average	13426250	3497393	13778977	3580324	13976706	3675466	13790347	3572429
Std. dev.	0.73%	1.58%	0.87%	1.61%	1.01%	2.26%	2.19%	2.84%
Overhead for 4 processors								
Overhead	2.66%	3.59%	0.79%	1.93%	4.31%	6.57%	2.42%	2.90%

Appendix B

SMP OKL4 and Locking

Details in this section were added to the thesis post-submission, for publication.

The version of the OKL4 microvisor assessed in this thesis was an SMP prototype for the NaviEngine, which has an ARMv7 ARM11 MPCore processor. It is important to note that the performance of some of the system calls shown in this report were either impacted by incomplete features or specific aspects of the NaviEngine platform. Many of the system calls that appear to scale poorly in these results have already been drastically improved, or would show much better scalability on a different platform. Further explanation follows.

When running on SMP hardware, the microvisor protects nearly all kernel operations with a global spinlock. The lock is acquired in the kernel's entry routines, and released immediately before returning to user level. All of the locking code is written in ARM assembly. The lock is a CLH queue lock (replaced in recent versions of OKL4 by a simpler ticket lock), so it is acquired in first-in first-out order.

The only operations that are not protected by the kernel lock are transitions between VCPU virtual user and virtual kernel modes, other than page faults. VCPU page faults do acquire the global lock, in order to walk the virtual page table.

The internal *make_consistent* mechanism is used to donate the lock to remote CPUs so they can perform simple maintenance operations. It works by having each CPU check a CPU-local event mailbox while spinning on the global lock. CPUs targeted by a make_consistent event are sent an IPI to force them to wait on the lock if they aren't doing so already. In the version of the microvisor used in this thesis, the IPI would cause the target CPU to acquire the lock and enter the kernel unnecessarily if it was not already spinning on the lock; recent versions of the kernel avoid entering the lock queue just to handle a make_consistent.

Some types of make_consistent event, such as cache flushes, can be sent to multiple remote CPUs simultaneously.

Operations that use make_consistent events include:

- Full data cache flushes on SMP ARMv7
- All cache and TLB flushes on ARM11 MPCore
- Flushing the lazily switched VFP/Neon state from a remote CPU's VFP registers
- Scheduler_seize (which interrupts a running thread on a remote CPU so its thread control block can be accessed safely)
- VIRQ delivery (in recent versions; this was formerly done using scheduler_seize)

B.1 Impact of make_consistent on system call scalability

In Section 6.3.3 we show a diagram that displays how various system calls scale. The worst of these system calls — A, B, C, E, J, K, O, P — all trigger a make_consistent event. The impact of this is especially bad on ARM11 MPCore, as it occurs on every cache and TLB flush. Since make_consistent has recently been taken out of the lock we can expect far better scalability with more recent versions of the OKL4 microvisor and on different platforms.

Bibliography

[ABB$^+$86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[Amd67] Gene M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings,*, pages 483–485, Atlantic City, NJ, USA, April 1967.

[ARJ97] James Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions of Computer Systems*, 15(2):134–165, 1997.

[BBD$^+$09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009. ACM.

[Bla90] David Black. Scheduling support for concurrency and parallelism in the Mach operating system. *Computer*, 23(5):35–43, 1990.

[BPS$^+$09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, May 2009.

[BWCC$^+$08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng

Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, USA, December 2008.

[CDL$^+$93] Eliseu Chaves, Prakash Das, Thomas Leblanc, Brian Marsh, and Michael Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: practice and experience*, 5(3):171–191, 1993.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly & Associates, Inc, 3rd edition, 2005.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, 2001.

[HL10] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, pages 19–24, New Delhi, India, August 2010.

[JAdG86] M. D. Janssens, J. K. Annot, and A. J. Van de Goor. Adapting UNIX for a multiprocessor environment. *Communications of the ACM: Computing Practises*, 29(9), September 1986.

[LES$^+$97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 28–31, Cape Cod, MA, USA, May 1997.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronisation on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, USA, January 1996.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *COMPUTING SYSTEMS*, 1:11–32, 1988.

[PS96] Sharon Perl and Richard L. Sites. Studies of Windows-NT performance using dynamic execution traces. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 169–183, October 1996.

[Rit93] Duncan Ritchie. The Raven kernel: a microkernel for shared memory multiprocessors. Technical Report TR 93-36, University of British Columbia, 1993.

[RN93] Duncan Ritchie and Gerald W. Neufeld. User level IPC and device management in the Raven kernel. In *Proceedings of the 2nd USENIX Workshop on Microkernels and other Kernel Architectures*, pages 111–125, San Diego, CA, USA, September 1993.

[Sch94] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison-Wesley, 1994.

[SCZM99] Peter Shirley, Ken Chiu, Kurt Zimmerman, and Steve Marschner. 252.eon SPEC CPU2000 benchmark. `http://www.spec.org/cpu2000/CINT2000/252.eon/docs/252.eon.html`, October 1999.

[Swa00] Bill Swartz. 300.twolf SPEC CPU2000 benchmark. `http://www.spec.org/cpu2000/CINT2000/300.twolf/docs/300.twolf.html`, January 2000.

[Uhl05] Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Karlsruhe, Germany, June 2005.

[USK93] Ron Unrau, Michael Stumm, and Orran Krieger. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9:105–134, 1993.

[YST$^+$07] Masayasu Yoshida, Takeshi Sugihara, Toshiaki Takahashi, Yasuhiko Koumoto, and Toshinori Ishihara. “naviengine 1,” system LSI for SMP-based car navigation systems. *NEC Technical Journal*, 2(4):35–39, 2007.