Scalability of Microkernel-Based Systems

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIERWISSENSCHAFTEN

von der Fakultät für Informatik
der Universität Fridericiana zu Karlsruhe (TH)
genehmigte

DISSERTATION

von

Volkmar Uhlig

aus Dresden

Tag der mündlichen Prüfung:	30.05.2005
Hauptreferent:	Prof. Dr. rer. nat. Gerhard Goos Universität Fridericiana zu Karlsruhe (TH)
Korreferent:	Prof. Dr. sc. tech. (ETH) Gernot Heiser University of New South Wales, Sydney, Australia
Karlsruhe:	15.06.2005

Abstract

Microkernel-based systems divide the operating system functionality into individual and isolated components. The system components are subject to application-class protection and isolation. This structuring method has a number of benefits, such as fault isolation between system components, safe extensibility, co-existence of different policies, and isolation between mutually distrusting components. However, such strict isolation limits the information flow between subsystems including information that is essential for performance and scalability in multiprocessor systems.

Semantically richer kernel abstractions scale at the cost of generality and minimality–two desired properties of a microkernel. I propose an architecture that allows for *dynamic adjustment* of scalability-relevant parameters in a general, flexible, and safe manner. I introduce isolation boundaries for microkernel resources and the system processors. The boundaries are controlled at user-level. Operating system components and applications can transform their semantic information into three basic parameters relevant for scalability: the involved processors (depending on their relation and interconnect), degree of concurrency, and groups of resources.

I developed a set of mechanisms that allow a kernel to:

1. efficiently track processors on a per-resource basis with support for very large number of processors,
2. dynamically and safely adjust lock primitives at runtime, including full deactivation of kernel locks in the case of no concurrency,
3. dynamically and safely adjust locking granularity at runtime,
4. provide a scalable translation-look-aside buffer (TLB) coherency algorithm that uses versions to minimize interprocessor interference for concurrent memory resource re-allocations, and
5. efficiently track and communicate resource usage in a component-based operating system.

Based on my architecture, it is possible to efficiently co-host multiple isolated, independent, and loosely coupled systems on larger multiprocessor systems, and also to fine-tune individual subsystems of a system that have different and potentially conflicting scalability and performance requirements.

I describe the application of my techniques to a real system: L4Ka::Pistachio, the latest variant of an L4 microkernel. L4Ka::Pistachio is used in a variety of research and industry projects. Introducing a new dimension to a system — parallelism of multiprocessors — naturally introduces new complexity and overheads. I evaluate my solutions by comparing with the most challenging competitor: the uniprocessor variant of the very same and highly optimized microkernel.

Zusammenfassung

Mikrokernbasierte Systeme teilen die Betriebssystemfunktionalität in unabhängige und isolierte Komponenten auf. Die Systemkomponenten unterliegen dabei denselben Isolations- und Schutzbedingungen wie normale Nutzeranwendungen. Solch eine Systemstruktur hat eine Vielzahl von Vorteilen, wie zum Beispiel Fehlerisolation zwischen Systemkomponenten, sichere Erweiterbarkeit, die Koexistenz mehrerer unterschiedlicher Systemrichtlinien und die strikte Isolation zwischen Komponenten, die sich gegenseitig mißtrauen. Gleichzeitig erzeugt strikte Isolation auch Barrieren für den Informationsfluß zwischen den individuellen Subsystemen; diese Informationen sind essentiell für die Performanz und die Skalierbarkeit in Multiprozessorsystemen.

Kernabstraktionen mit semantisch höherem Gehalt skalieren auf Kosten der Allgemeinheit und der Minimalität, zwei erwünschte Eigenschaften von Mikrokernen. In dieser Arbeit wird eine Architektur vorgestellt, die es erlaubt, die für die Skalierbarkeit relevanten Parameter generisch, flexibel, und sicher dynamisch anzupassen. Es werden Isolationsschranken für die Mikrokernresourcen und Systemprozessoren eingeführt, welche unter der Kontrolle von Nutzerapplikationen stehen. Die Betriebssystemkomponenten und Anwendungen können das ihnen zur Verfügung stehende semantische Wissen in die folgenden drei skalierbarkeitsrelevanten Basisparameter umwandeln: die involvierten Prozessoren (abhängig von den Prozessorbeziehungen und dem Speichersubsystem), den Grad der Parallelität und Ressourcengruppierungen.

Es wurden die folgenden Methoden und Mechanismen entwickelt:

1. eine effiziente Methode zur Speicherung und Auswertung der zu einer Ressource zugehörigen und relevanten Prozessoren,

2. ein dynamisches und sicheres Synchronisationsprimitiv, welches zur Laufzeit angepaßt werden kann (dies beinhaltet die vollständige Deaktivierung von Kernsperren für den Fall, daß keine Parallelität vorhanden ist),

3. die dynamische und sichere Anpassung der Granularität von Sperren zur Laufzeit,

4. ein skalierbarer *translation-look-aside buffer* (TLB) Kohärenzalgorithmus, der zur Vermeidung von wechselseitigen Beeinflussungen von Prozessoren ein Versionsschema nutzt, sowie

5. ein Mechanismus zur effizienten Ermittlung und Weiterleitung von Ressourcennutzungsinformationen in einem komponentenbasierten Betriebssystem.

Basierend auf dieser Architektur ist es möglich, mehrere unabhängige, isolierte, und lose verbundene Systeme gleichzeitig auf einem Mehrprozessorsystem zu betreiben. Desweiteren ermöglicht die Architektur, einzelne Subsysteme

individuell feinabzustimmen. Dies ist selbst dann möglich, wenn Subsysteme unterschiedliche oder sogar widersprüchliche Skalierbarkeits- und Performanceanforderungen haben.

Die Techniken werden an einem real existenten System exemplarisch evaluiert: dem L4Ka::Pistachio Mikrokern, der die neueste L4-Version darstellt. L4Ka::Pistachio wird aktiv in einer Reihe von Forschungs- und Industrieprojekten eingesetzt. Die Einführung der neuen Dimension *Parallelität von Prozessoren* erhöht sowohl die Komplexität als auch die Kosten. Die Effizienz der vorgestellten Lösungen werden daher an dem größten Konkurrenten gemessen: der Uniprozessorvariante desselben hochoptimierten Mikrokerns.

Acknowledgements

First, I would like to thank my supervisor, Prof. Gerhard Goos. After the loss of Jochen Liedtke, he not only ensured the survival of our group but rather its blossoming. It took me a while to realize the wisdom behind many of his demanding requests and I am extremely thankful for his patience and guidance over the last few years. I am particularly thankful for many of his advices in areas that were not related to this work or even computer science.

Jochen Liedtke, who passed away in 2001, was most influential on my interest in research in general and operating systems in particular. I am grateful to him in many ways: he was a great researcher, excellent teacher, mentor, and friend. This work would not exist without his groundbreaking achievements, the astonishing intellect he had, and his ability to help all of the people around him thrive.

I want to specifically thank Joshua LeVasseur, who is a good friend and colleague. With him, it was not only possible to achieve excellent research results, but also to have a life in Karlsruhe. I hope we will continue to work together over the coming years. Thanks also to Susanne for her endless supply of excellent food and many fun hours.

Thanks to Jan Stöß for his efforts in our joined projects, including but not limited to multiprocessor L4. Andreas Haeberlen and Curt Cramer have both been influential in many ways; I enjoyed the constant technical and political discussions. I still owe Andy a two-lettered rubber stamp. I had endless hours of brainstorming and discussions with Lars Reuther and many of my research ideas (including those in this thesis) were born with him on the other end of the phone line.

I am thankful to the people of the System Research Lab of Intel MTL, in particular Sebastian Schönberg and Rich Uhlig. It was a wonderful time working in such an inspiring environment and getting deep insights into the hardware architecture. Rich is an excellent manager and good friend and was supportive in many ways. Intel supported this research with a generous research grant, which financed the required multiprocessor hardware and also supported me during the last three months of my thesis. I really enjoyed the productive and intellectually challenging work with Sebastian, but also our regular Wednesday evenings at Imbri Hall.

I thank Gernot Heiser for his support, for reading initial drafts of this thesis, and coming over from Australia for the defense. Although we have had many flame wars on internal and external mailing lists, it is always a joy to work with him. I also want to thank Gernot for offering me a PhD position in Sydney when things got rough.

I want to thank Julie Fast for editing the final version of this thesis, and also for her support and encouragement. I also have to thank Monika Schlechte, my sister, and my parents for their constant support on this interesting journey. Monika was probably most influential on the directions of my life including starting a research career. Jan Langert and Michaela Friedrich were of great support during the hardest years of my PhD studies in 2002. I want to thank them for their patience, the uncountable hours on the phone, and their very helpful advice.

I have to thank the people of the System Architecture Group for making L4 and L4Ka::Pistachio such a great success. I want to specifically mention Espen Skoglund and Uwe Dannowski who helped make V4 happen, James McCuller for the most amazing IT environment ever, and Gerd Liefländer for shouldering the primary teaching load and thus enabling our research results (even though Gerd was always skeptical that I will finish on time). Without those people this work would not have been possible. I also want to thank Frank Bellosa, who is now the new head of our group. He has not only been very supportive in the short period we have been working together, but also gave great advice during my interviewing time.

Last, but not least I have to thank the whole L4 community for the insightful discussions on microkernel design. We still have a long way to go and I hope it will be as much fun in the upcoming years.

Contents

Chapter 1

Introduction

In the last years two important trends are changing the systems area: First, performance scaling by increasing processor frequency is reaching the point where further frequency increases are uneconomical. The current leakage that is inherent in small structure size of today's high-frequency processors results in massive energy dissipation that is emitted as heat. In order to further increase compute power, processor development is switching the focus to increasing parallelism. All primary processor vendors are providing or announcing systems with a large number of tightly coupled processors, that have previously been only available in mainframe-class systems.

Second, the increasing variety of usage cases of computers as multi-purpose devices is setting new demands on operating systems which remain unfulfilled by traditional operating systems, such as UNIX and Windows. These new demands require radical new system designs, and cannot be achieved by simply extending the existing systems with new features. Examples of these requirements are higher level of security and confidentiality of critical information, real-time support, and safe sharing of hardware resources in consolidated server environments of potentially untrusted clients.

I argue, that microkernels present a viable alternative for structuring operating systems that fulfill these new system requirements *if they scale on large multiprocessor systems*.

Scalability of operating systems for shared-memory multiprocessors is a complex problem that requires careful design of the complete software stack. The complexity stems from the structure of the underlying hardware architecture. With an increasing number of processors resource access latencies are non-uniform which has to be considered by the operating system. Naïve synchronization and resource allocation schemes that are efficient in smaller multiprocessor systems often lead to resource contention, starvation, high overhead, and ultimately to system saturation on more sophisticated hardware.

The operating system is an omnipotent software layer on top of the hardware and therefore a crucial contributor to overall scalability. It manages and multiplexes

all hardware resources and mediates them between the competing applications. Depending on resource usage patterns, the degree of resource sharing, object scope, and object granularity, the operating system selects the most appropriate and efficient synchronization algorithm, and thereby minimizes overhead and maximizes scalability. The choice for the optimal algorithm requires detailed semantic information about system subjects and objects. For example, the relevant literature differentiates between as many as ten different synchronization schemes.

In microkernel-based systems, the microkernel is only a very thin software layer that multiplexes the hardware between isolated or cooperating entities in a secure, safe, and extensible manner. The managed resources are limited to basic memory pages, processor resources, and processor time. All high-level operating system abstractions, such as file systems, network stacks, and device drivers are implemented as unprivileged user-level servers.

This system structure drastically reduces the complexity of the kernel itself and previous research validates that it is possible to construct real-time [53] and secure systems [90] on a microkernel. However, moving operating system constructs out of the kernel also eliminates detailed semantic information that is required to achieve optimal scalability. The microkernel lacks the necessary information to make an educated decision on the most efficient strategy.

Previous research on multiprocessor microkernel systems therefore compromised either on the scalability aspect (by choosing one specific synchronization policy), or on the strict separation of mechanism and policy and blurred the line between kernel and applications. The latter approach looses the desired properties of a microkernel.

In this thesis I present solutions that allow for an uncompromised microkernel design while achieving excellent scalability with low overhead. My solutions comprise four areas: I developed a space and time efficient tracking scheme for parallelism allowing every kernel object to be tagged with a processor mask. Using the tracking mask, the kernel derives the potential concurrency on kernel objects and adjust its synchronization scheme.

I developed *dynamic locks* that can be safely enabled and disabled at runtime thus eliminating synchronization overhead while still guaranteeing functional correctness in the kernel. Based on the dynamic locking scheme, I derived a method for dynamically adjusting the *lock granularity* for divisible objects. This provides applications with control over lock granularity in the microkernel. Applications can dynamically choose between coarse-grain locking for lower runtime overhead or fine-grain locking for a higher level of parallelism.

Memory is a primary resource managed at application level but enforced by the microkernel. Besides synchronization on kernel meta data, the microkernel has to enforce coherency of the translation look-aside buffer (TLB). I present an algorithm that decouples memory permission updates from outstanding TLB updates. With this scheme, multiple processors can manipulate permissions to memory objects in parallel while still minimizing the overhead for TLB coherency updates.

Finally, I present a low-overhead event logging mechanism that transfers

scheduling-relevant resource usage data between system components. The mechanism uses memory as a high-bandwidth and low-overhead transport mechanism and thereby only induces a marginal runtime overhead. It enables application-level schedulers to efficiently manage and allocate kernel resources, such as allocating threads to processors and also managing the per-processor kernel memory pools.

I describe the application of my techniques to a real system: L4Ka::Pistachio, the latest variant of an L4 microkernel. I played a significant role in the design and development of L4Ka::Pistachio. At this time, it supports nine different hardware architectures and is used in a number of industry and research projects. Introducing a new dimension to a system — parallelism of multiprocessors — naturally introduces new complexity and overheads. I evaluate my solutions by comparing with the most challenging competitor: the uniprocessor variant of the very same and highly optimized microkernel.

I see my contribution as extending minimalistic microkernel design, epitomized by L4, to the important domain of multiprocessors. However, the solutions are not restricted to microkernels, but are also applicable in the area of virtual machines, monolithic operating systems and even applications.

Organization

This thesis is structured as follows: In Chapter 2, I define common terms and principles required for the following sections. In Chapter 3, I evaluate related work on synchronization, scalability of multiprocessor operating systems in general, and microkernel-based systems in particular.

In Chapter 4, I develop my principles for adaptive synchronization primitives for the kernel. In Section 4.1, I discuss the structural differences of microkernel-based systems as compared to traditional operating systems for large multiprocessor systems. Then, I develop my resource tracking scheme (Section 4.2) which forms the foundation for the dynamic locking scheme described in Section 4.3. In Section 4.4, I describe the TLB coherency tracking algorithms that decouples memory permission management and TLB coherency updates. Finally, in Section 4.5, I develop the event-logging mechanism that enables the microkernel and isolated system components to efficiently exchange resource usage information that is required for user-level scheduling and multiprocessor resource allocation policies.

Chapter 5 shows how I applied my methodology to L4Ka::Pistachio, an L4 microkernel. In Section 5.1, I give a general overview of L4 primitives required for the later development. In Section 5.2, I specify functional and performance requirements and define design goals and non-goals. I then introduce the multiprocessor extensions to the uniprocessor kernel abstractions. In Section 5.3, I describe the implications of parallelism to L4's most performance-critical primitive: interprocess communication (IPC). I use dynamic locks for IPC to eliminate the synchronization overhead for the most important common cases. Section 5.4 details scalable user-level memory management applying dynamic locking and the

TLB coherency scheme to L4. Finally, in Section 5.5, I show how event-tracing can be used to manage kernel memory and threads via a user-level scheduler.

In Chapter 6, I evaluate the performance of my design for L4Ka::Pistachio. After detailing the peculiarities of my evaluation platform in Section 6.1, I evaluate the performance and scalability of the multiprocessor IPC primitive on the same and different processor (Section 6.2), the event logging scheme (Section 6.3), and user-level memory management (Section 6.4).

Chapter 7 includes a summary and suggestions for future work.

Chapter 2

Facts and Notation

In this chapter I introduce common terminology and principles of operating systems and multiprocessor hardware that is used throughout the later chapters in this dissertation.

Operating System and Microkernels

An operating system (OS) is a program (or a set of programs), which mediates access to the basic computing resources provided by the underlying hardware. Most operating systems create an environment in which an application can run safely and without interference from other applications. In addition, many OSes provide the application with an abstract, machine independent interface to hardware resources that is portable across different platforms.

There are two popular views of an operating system: The operating system as a resource manager or the operating system as an abstract virtual machine.

The view of a resource manager has the operating system acting as an arbiter for system resources. These resources include disks, networks, processors, time and others. The resources are shared among the various applications depending on individual applications' requirements, security demands, and priority.

An alternative view of operating systems is that of an abstract virtual machine. Each virtual machine provides a level of abstraction that hides most of the idiosyncrasies of lower-level machines. A virtual machine presents a complete interface to the user of that machine. This principle can be applied recursively.

An operating system provides an interface to its applications to enhance the underlying hardware capabilities. This interface is more portable, provides protection among competing applications, and has a higher level of abstraction than bare hardware. Briefly, operating systems typically provide services in the following areas: (*i*) program creation, (*ii*) program execution, (*iii*) access to I/O devices, (*iv*) controlled access to devices, (*v*) error detection and response, and (*vi*) accounting.

A number of different architectural organizations are possible for an operating system; the two most relevant structures are *monolithic* and *client-server*. In the monolithic system, all OS functions are integrated into a single system image.

The advantage is good performance with the disadvantage of high complexity. In monolithic systems, all subsystems share the same protection domain and faults in one subsystem can propagate into others. Client-server systems employ a *microkernel*, that mediates resource access to the bare hardware. The OS subsystems are implemented as applications that run in their own address space. The microkernel provides inter-process communication for interaction between the subsystems. Microkernels provide significantly better fault isolation because subsystems are confined to their own protection domain.

Multiprocessor Systems

In multiprocessor systems, multiple connected processors operate in parallel. Depending on the processor interconnect, the systems are differentiated in (i) *clusters* (or multicomputers) and (ii) *shared memory multiprocessors*. In clusters, each processor has its dedicated memory and is an autonomous computer; the computers communicate via a network. Shared memory multiprocessors have common memory for code and data and communication between processors may take place via shared memory.

One general classification of shared memory multiprocessors is based on how processes are assigned to processors. The two fundamental approaches are *master/slave* and *symmetric*. For master/slave, the operating system always runs on a particular processor and the other processors execute applications. In *symmetric multiprocessor systems* (SMP), the operating system can execute on any processor.

For a small number of processors, multiprocessor systems commonly use a *shared memory bus* that connects all processors and the memory. The shared bus structure limits the overall memory bandwidth, because only one processor can access the bus at a time. With an increasing number of processors, the shared bus becomes a performance bottleneck. Large scale systems therefore use either multiple interconnected buses or per-processor memory. The buses are interconnected via memory routers. These systems, however, exhibit different access latencies for memory that is local to the accessing processor compared to remote memory. The *non-uniform memory access* property gave the systems their name: NUMA.

With the increasingly high integration, two other multiprocessor structures become more common: *simultaneous multithreading* (SMT) and *chip-level multiprocessing* (CMP). SMT uses the multiple-issue-per-instruction features of modern superscalar processors to hide latencies of resource stalls, such as cache misses. SMT processors [108] provide multiple threads of execution that share the functional units of one processor. On resource stalls the processor automatically switches to another thread. The primary goal of SMT is to increase the utilization of functional units of one processors.

CMP is an SMP implemented on a single chip. Multiple processor cores typically share a common second- or third-level cache and interconnect. CMPs are also often referred to as multicore processors.

Caches

Cache memory is intended to give memory speed approaching that of the fastest memory available, and at the same time provide a large memory size at the price of less expensive memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine whether the word is in the cache. If so, the word is delivered to the processor, otherwise a block of memory is fetched from main memory into the cache.

A consistency problem arises in multiprocessor systems, where each processor has its own cache. When one processor modifies a datum, the other processors' caches may still contain the old value, which is now incorrect. *Cache coherency mechanisms* that are implemented in hardware transparently update the processor caches. In bus-based systems processors can *snoop* the memory bus for addresses that they have cached. When a write operation is observed to a location that a cache has a copy of, the cache controller invalidates its own copy. Various models and protocols have been devised for maintaining cache coherency (e.g., MESI, MSI, MOSI, and MOESI).

In cache-coherent NUMA systems (ccNUMA), processors have no common memory bus prohibiting cache snooping for cache coherency. Instead, ccNUMA systems use directory tables to keep track of which processors cache a specific word. On modification, the hardware sends directed update and invalidate messages to the processors. This scheme has a significantly higher overhead compared to snoopy caches and requires specific attention when designing a system.

Memory Management Unit

The memory management unit (MMU) of a processor is responsible for handling memory accesses requested by the CPU. Among others, the MMU translates virtual address to physical addresses (to implement virtual memory), it enforces memory protection, and controls the caching strategy. The virtual address space is divided into pages of a size of 2^N, usually a few KBytes. When accessing memory, the lower N bits remain unchanged while the upper bits select the (virtual) page number. The page number is used to index into a *page table*, that contains the translation from virtual address to the physical memory page.

To reduce the translation latency, processors cache page table entries in the *translation look-aside buffer* (TLB) [66]. TLBs work similarly to normal caches and contain the page table entries most recently used. Given a virtual address, the processor will first inspect the TLB, and if the page table entry is present (i.e., a TLB hit) the physical address is retrieved. Otherwise, the processor examines the page table and if a valid entry is found the entry is loaded into the TLB. If the page table contains an invalid entry, the processor signals a page fault exception to the operating system.

Similar to memory caches, TLBs need to be kept coherent between multiple processors. While cache coherence for memory is implemented in hardware, en-

forcement of TLB coherence is left to the operating system. When updating a page table entry, the operating system has to specifically issue a *TLB invalidation* instruction. With very few exceptions, architectures require explicit invalidation of TLB entries on remote processors. The updating processor sends an inter-processor interrupt (IPI) to the remote processors, which then invoke the TLB invalidation instruction updating their respective TLB. This remote invalidation is commonly referred to as *TLB shoot-down*.

Chapter 3

Related Work

Related work to this thesis can be classified into the following four areas:

Scalable systems. My thesis proposes a number of mechanisms that control and adjust kernel algorithms for resource allocation and management. In Section 3.1 I discuss the general problem of multiprocessor scalability. I review previous work that addresses the problem of operating system scalability, discuss the relevant findings and relate them to my work.

Locks and synchronization. Locks are an important primitive for mutual exclusion in multiprocessor systems but also a main source for limited scalability and overhead. My thesis proposes an adaptable locking mechanism. In Section 3.2 I describe the scalability problem of locks and other approaches that address lock overhead and lock contention in scalable systems.

Partitioning. I use resource partitioning to isolate independent subsystems and thereby avoid potential interference. In Section 3.3 I review other approaches — software and hardware — that use resource partitioning to achieve better scalability.

Microkernel-based systems. This thesis investigates scalability of microkernels. As the disappointing performance results of early approaches have shown, microkernels require careful design and implementation. In Section 3.4 I describe the state of the art in microkernel-based systems laying the ground for many design decisions, but also discuss other related work in the field of microkernels on multiprocessors.

3.1 Scalable Systems

Multiprocessor scalability is addressed at various levels of the system stack: the hardware level, the operating system, and the application and algorithmic level.

At the hardware level, a higher degree of circuit integration and the limitations of frequency scaling puts a stronger focus on parallelism. Most microprocessor

vendors ship now or have announced multicore versions of their processors, and offer multiprocessors based on a single shared bus. Single bus systems, however, are not scalable beyond a small number of processors. Because of this, the memory bus becomes the system's performance bottleneck.

The first scalable shared-memory machines are based on Omega 0 networks [101], such as the NYU Ultra-Computer [3], the IBM RP3 [89], and the BBN Butterfly [28]. In these systems, each processor has locally accessible memory, but is able to access the memory of remote processors via the memory interconnect. Accesses to remote memory have a higher latency than to local memory. The non-uniformity for memory accesses gives these machines their name: *NUMA*. Alternative interconnect structures are ring or grid topologies, as used by the Stanford DASH [68] and KSR [24] architectures, and the HECTOR [118] multiprocessor.

A critical design issue of all shared memory architectures is the cache coherency scheme. Each processor in a multiprocessor system has its own cache; hence data can appear in multiple copies among the various caches. Cache coherence ensures that all processors have identical copies of a datum. In shared bus systems, snoopy caches [11] provide a very simple method for cache coherence, whereas NUMA systems employ directory-based cache-coherency approaches [50]. In this scheme, a directory keeps track of which processors have cached a given memory block. When a processor wishes to write into that block, the directory sends point-to-point messages to processors with an updated copy, thus invalidating all other copies.

Cache-coherency updates and cache-migration costs are the prime limiting factors for scalability and need to be considered for software and system construction. The overhead for cache-coherency updates has implications on all areas of multiprocessor systems, including the selection of synchronization primitives (such as locks), the overhead for data sharing between processors, and for memory allocations in order to reduce the cost for memory accesses.

The metric for software scalability is speedup. According to Amdahl's law [4], the maximum speedup that parallelism can provide is bounded by the inverse of the fraction that represents the serial portion of the task. Theoretical parallel computing models, such as PRAM [38, 87] and LogP [31], provide abstractions of hardware machines in a portable high-level manner. For example, LogP incorporates four parameters into its model: communication latency, communication overhead, the gap between consecutive messages, and the number of processors. However, LogP and PRAM assume that they can distribute the workload across processors and model a strategy for load distribution. This strategy is not applicable to operating systems and therefore theoretical models consider the operating system as a given extension of the hardware platform.

3.1.1 Operating System Scalability

Unrau's thesis [113] is probably the first that addresses operating system scalability in a new and systematic manner. Instead of addressing the algorithmic speedup, he

proposes the concept of the operating system as a *service center*. His work bases the scalability evaluation on queuing theory [65] with the three fundamental performance metrics: throughput, utilization, and response time. Based on his analysis, Unrau derives the following three fundamental design criteria for a scalable operating system that are quantified as runtime overhead of an operating system function.

Preserving parallelism: *The operating system must preserve the parallelism afforded by the application.* If several threads of an executing application request independent operating system services, then they must be serviced in parallel. Otherwise the OS becomes a bottleneck limiting scalability and application speedup.

Bounded overhead: *The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors.* If the overhead of each service call increases with the number of processors, the system will ultimately saturate.

Preserving locality: *The operating system must preserve the locality of the application.* It is important to consider the memory access locality in large-scale systems, because many large scale multiprocessors have non-uniform memory access times, where the cost of accessing memory is a function of the distance among accessing processors, and because cache consistency incurs higher overhead in large systems.

While these three design principles are a necessary requirement for scalable systems, they are not sufficient for the following reason: Unrau's requirements only address the scalability of a *given system*. This approach leaves two important aspects unaddressed: *overhead* and *isolation*.

According to Unrau's first and second design rule, a system is still considered scalable if the actual execution time of a function is significantly higher than the achievable minimum. The problem thereby is to determine the minimally achievable overhead, which depends on the specific system scenario and on the number of processors.

In order to determine the minimal overhead, I propose to incorporate an additional aspect: the *isolation property* of an operation. This proposal is based on the following observation from hardware partitioning: Hardware partitioning divides a large multiprocessor system into a set of independent smaller systems. The individual subsystems are fully isolated and behave (almost) like a physically partitioned multiprocessor system. Most importantly, the isolated subsystems *preserve* their performance and scalability properties. Hence, the achievable minimum overhead for an operation, which is executed on (or affects) only a subset of the processors of the system, should be identical to the overhead of a system of the size of the subsystem.

The overhead of an operation is a function of its isolation boundaries; the boundaries are represented by the processors affected by the operation. I propose

a fourth design rule that addresses the overhead of a system function in relation to its isolation boundaries:

Preserving isolation: *The operating system must preserve the isolation properties of an application.* When applications are restricted to a subset of processors, the overhead for a system operation should be identical to the overhead of a system with the size of the processor subset.

An implication of that requirement is that the cost for operations which are restricted to a single processor should also have the overhead of a single-processor system.

A system with more processors has an inherently higher management overhead. This overhead includes memory resources that need to be allocated and more complex algorithms in order to accommodate hardware peculiarities which do not exist in smaller hardware configurations. Therefore, it is hard to strictly follow the postulated isolation requirement. However, in some cases it is possible to trade memory footprint against less performance for the uncommon case.

Unrau's third design rule addresses the special case of NUMA access overhead. More generally, the operating system should use memory that is isolated to one (or a subset) of the system's processors.

3.1.2 Clustering

Hierarchical clustering is a way to structure shared-memory multiprocessor operating systems for scalability [113,114]. The basic unit of structuring is a cluster. A cluster provides the functionality of an efficient, small-scale SMP that comprises a small number of processors only. On larger systems, multiple clusters are instantiated such that each cluster manages a group of neighboring processing modules. A processing module consists of a small number of processors, local memory, and the memory interconnect. Major system services are replicated to each cluster so that independent requests can be handled locally. In a later paper, Gamsa et al. [41] list a number of problems with this approach, including poor locality, increased complexity, and difficulties to support customized policies.

Clustered objects [7,41] (as used in Tornado and K42) extend the idea of a clustered system structure to the OS object granularity (i.e., a C++ object instance). A clustered object presents the illusion of a single object but is actually composed of multiple object instances. Each individual instance handles calls from a subset of the processors and the object instance presents the collective whole for all processors. The specific implementation of clustered object instance can differ to reflect the degree of parallelism and access patterns. The key to the implementation is the use of a per-processor translation table that contains a reference to a handling method. The indirection table allows the creation of object representatives on demand by installing a default fault handler in the table. When an object is first referenced on a processor, the default handler catches the access attempt and instantiates a local representative.

Clustered objects are a powerful method for runtime adaptation. In particular, it is possible to dynamically replace object code and the object's synchronization models via the indirection table. The faulting scheme, however, creates the problem that it only supports a forward scheme. The per-processor representatives are established on access fault. There is no similar event for the destruction of representatives and a fall-back to a less-scalable scheme that may incur a lower overhead. While the authors describe a garbage collection mechanism very similar to read-copy-update, there is no equivalent for a down-sizing (or de-parallelization) of a clustered object. While this may not impose a problem for short-lived kernel objects, it is critical for long-lived objects with significantly changing access patterns, as seen in microkernels with user-level resource management.

3.2 Locks and Synchronization

Locks provide individual processors with exclusive access to shared data and critical sections of code. The most common locks for in-kernel synchronization are *spin locks*, where a shared memory location stores the lock state (i.e., taken or free). In order to acquire the lock, the processor atomically replaces the content of the lock variable with the value that denotes a taken state. If the previous value was *free*, the lock was acquired successfully, or otherwise the processor retries in a loop.

The overhead for locks falls into three different categories: (*i*) the runtime overhead for the lock primitive itself, (*ii*) the cache footprint for the lock, and (*iii*) utilization of memory bus bandwidth.

Locks are implemented with atomic processor primitives. A variety of alternative atomic primitives have been proposed and implemented on shared memory architectures in order to minimize locking overheads. Many of those special primitives support one particular synchronization operation [45, 59, 68]. Although special-purpose primitives have advantages in certain scenarios, common processor architectures support a set of general-purpose primitives, such as FETCH&Φ [46], COMPARE&SWAP [25], and the pair of LOADLINKED–STORECONDITIONAL [61].

- The FETCH&Φ primitive takes two parameters: the address of the destination and a value parameter. The primitive atomically reads the value from the destination, computes the new value as $\Phi(\text{original value}, \text{parameter})$, stores it, and returns the original value.

  Example primitives are TEST&SET, FETCH&ADD, and FETCH&STORE[1].

- COMPARE&SWAP takes three parameters: the address of the destination, an expected value and a new value. If the original value at the destination

[1] FETCH&STORE is also known as EXCHANGE that atomically replaces a register value with a memory location. It is the atomic operation with the lowest overhead for IA32.

address is identical to the expected value, it gets replaced atomically with the new value. The return value indicates success or failure.

- The pair LOADLINKED–STORECONDITIONAL must be used together to read, modify, and write a shared location. LOADLINKED returns the value stored at the shared location and at the same time sets a reservation associated with processor and location. The reservation remains valid until another processor accesses the recorded cache line. STORECONDITIONAL checks the reservation; if valid it writes the new value and returns success, otherwise failure. The reservation is commonly implemented in the processor cache and invalidation can be easily embedded in the cache snooping protocol. When a STORECONDITIONAL fails it fails locally and does not cause any bus traffic.

3.2.1 Locks: Software Approaches

The naïve use of atomic operations can induce significant overhead in larger multiprocessor systems. Processors that compete for a lock create memory bus traffic due to the cache coherency protocol. Alternative locking schemes, such as Mellor-Crummey Scott locks (MCS locks) [82], reduce the overhead with locking queues and employ spinning on local memory. Local spinning reduces the overhead of the cache coherence protocol from all competing processors to one.

The optimal synchronization scheme depends on the peculiarities of the scenario, such as the level of contention, the memory interconnect of competing processors, and access frequency and pattern. Less concurrency and less complex memory interconnects (such as simple snoopy caches of SMP systems) also require less complex lock mechanisms and code paths, thus reducing runtime overhead and cache footprint.

Overhead and complexity increases when locks are contended and lock attempts cross NUMA node boundaries. However, no single best locking scheme exists. Anderson [6] observed that the choice between spin-locks and MCS queue locks depends on the level of lock contention. Lim [76] proposes dynamic runtime adaptation between both lock protocols. Depending on the degree of lock contention the code adapts the lock protocol. For low contention cases (with less than four processors) the scheme uses spin locks with the significantly lower runtime overhead, whereas in high contention cases the more expensive, but better scalable, MCS locks are used.

McKenney [79] argues that the choice for the best locking strategy depends on a large number of factors, such as duration of critical section, read-to-modify ratio, contention, and complexity. He differentiates as much as ten alternative design patterns for choosing an optimal locking strategy.

Unrau et al. [115] address the problem of overhead vs. parallelism for fine granular locks. They propose a hybrid coarse-grain/fine-grain locking strategy that has the low latency and space overhead of a coarse grain locking strategy while having

the high concurrency of fine-grain locks. The idea is to implement fine-granular locks using non-atomic memory operations and protect the lock data structures with a single coarse lock. The coarse lock must be held in order to acquire the fine granular locks, which then can be implemented as a simple reserve bit. When spinning on the lock, only the reserve bit is tested and does not require an acquisition of the coarse lock.

The scheme has two limitations. First, the coarse lock still has to be taken by *all processors* and thus needs to be migrated between the different processor caches. Real fine granular locks can eliminate that inter-dependency and overhead. Second, in order to reduce contention on the coarse lock, processors spin on the reserve bit of the object. Only when the reserve bit gets cleared do the processors retry the lock acquisition via the coarse lock. That scheme puts restrictions on object destruction and prohibits free memory reuse. A lock that is embedded in an object which is released and re-allocated may never be cleared. Processors that are spinning on such a lock will spin infinitely. The authors solve this problem by introducing object classes for memory in order to guarantee that the particular reserve bit reaches an unlocked state.

3.2.2 Locks: Hardware Approaches

In addition to software-based approaches, research proposes special hardware support in order to minimize the lock overhead.

Goodman et al. proposed the Queue-On-Lock-Bit primitive (QOLB) [45] that was also the first proposal for a distributed, queue-based locking scheme. QOLB maintains a hardware queue of waiting processors in the cache so that processors can spin locally. On lock release the first processor in the queue receives a specially formed cache-coherency packet; the other processors remain spinning. Kägi et al. [63] compares the throughput of TEST&SET locks, TEST&TEST&SET locks [95], MCS locks, LH locks [77], M locks [77], and QOLB locks. The QOLB locks outperform all other lock mechanisms, however, the locks rely on support of the special QOLB primitive. The simple TEST&SET and TEST&TEST&SET locks performs well under low contention but throughput degenerates quickly for more than four processors.

Rajwar et al. [92] reduce the runtime overhead for locks via *speculative lock elision* (SLE). SLE is a micro-architectural technique similar to speculative execution. The processor dynamically identifies synchronization operations, predicts them as unnecessary, and elides them. By removing these operations, the program behaves as if synchronization were not present in the program. Conflicts that would violate the correctness due to the missing synchronization primitives, are detected via the cache-coherency mechanisms and without executing the actual synchronization operations. Safe dynamic lock removal exploits the properties of locks and critical sections as they are commonly implemented. If data is not concurrently accessed between the lock acquisition and release operations, both operations can be elided. Data written within the critical section is buffered in an intermediate

buffer while monitoring the lock variable. If the operation completes without a violation of atomicity, the buffer is written back. Otherwise the operation is restarted with a normal lock.

Speculative lock elision addresses the problem of the high overhead for lock primitives in cases of no concurrency. While SLE is a generic and transparent solution it has limitations. SLE requires an extension of the hardware microarchitecture and is thus not applicable to currently available hardware architectures. The size of the critical section is further limited by the size of the write buffer. Since the processor monitors the cache line of the lock, SLE induces cache and memory footprint for the lock variable. Furthermore, nesting of locks requires multiple locks being monitored concurrently.

3.2.3 Lock-free Synchronization

An alternative to lock-based synchronization is lock-free synchronization. Critical code sections are designed such that they prepare the results out of line and then try to commit them using an atomic update instruction such as COMPARE&SWAP. The most prominent operating systems using lock-free synchronization are the Cache kernel [48], Synthesis [78], and Fiasco [58], which is an L4 microkernel variant. (A more detailed discussion of Fiasco follows in Section 3.4.2.)

The Cache kernel and Synthesis run on architectures with a COMPARE&SWAP2 instruction (Motorola 68K). The authors report lock-free synchronization as a viable alternative, however, they do not address the performance implication due to frequent atomic operations or lock contention. For example, the SYNTHESIS kernel was only tested on a two-way machine with a shared memory bus and the CACHE kernel on a four-way machine. The authors argue that the overhead induced by lock-free synchronization in number of *instructions* is minimal. However, the overhead of the COMPARE&SWAP2 operation is reported to be as high as 114 cycles. The reported overhead for a variety of common kernel operations using lock-free synchronization vs. unsynchronized operations is between 50 up to 350 percent.

Lock-free synchronization via atomic operations further eliminates the possibility for *critical-section fusing* [79], where multiple critical sections are protected by a single lock. Hence, while a single lock may be highly sufficient, scalable, and induce a very low overhead, the lock-free synchronization scheme adds the overhead of an atomic COMPARE&SWAP2 to every critical section.

3.2.4 Message-based Synchronization

Chaves et al. [26] discuss alternative in-kernel synchronization schemes for large-scale shared memory multiprocessors. The authors distinguish between three primary synchronization mechanisms for kernel–kernel communication:

Remote memory access: The operation executes on processor *i* reading and writing processor *j*'s memory as necessary.

Remote invocation: Remote invocation is based on sending a message from processor *i* to processor *j* asking *j* to perform the operation on behalf of *i*.

Bulk data transfer: The kernel moves the data required by the operation from processor *j* to *i*, where it is inspected or modified and possibly copied back.

The work discusses the performance trade-offs for the different synchronization schemes. The authors conclude that the choice between remote invocation and remote access is highly dependent on the cost of the remote invocation mechanism, the cost of the atomic operations used for synchronization, and the ratio of remote to local memory access time. Although, coming to this conclusion, the authors do not discuss methods on *how* to resolve the open problem.

A number of operating systems use in-kernel messaging for synchronization: most prominent are the DragonFly BSD project [1], which bases most in-kernel synchronization on inter-processor interrupts, and Tornado [41].

I am not aware of any previous work which provides the choice for dynamic selection of message *and* lock based synchronization in one kernel.

3.2.5 Read-Copy Update

For read-mostly data structures, performance can be greatly improved by using asymmetric locking primitives that provide reduced overhead for read-side accesses in exchange for more expensive write-side accesses. *Read-copy update* (RCU) [80, 81] takes the idea to the extreme, permitting read-side accesses with no locking or synchronization. This means that updates do not block reads, so that a read-side access that completes shortly after an update can return old data.

Data structures in parallel systems cannot be considered stable unless a particular update policy is followed, such as holding locks to data. After the locks are released, the system cannot make any prior-knowledge assumptions about the state of the data which was protected by the locks. Thus, if a thread does hold no lock, it cannot make any assumption about *any* data structure that is protected by any lock. When a thread holds no locks, it is in a quiescent state with respect to any lock-protected data.

The fundamental idea of read-copy update is the ability to determine when all threads have passed through a quiescent state since a particular point in time. Afterward, it is guaranteed that the threads see the effects of all changes made prior to the interval. This guarantee significantly simplifies many locking algorithms and eliminates many existence locks.

The processors signal the event of passing a quiescent state by circulating a token. When the processor that currently possesses the token reaches a quiescent state, it passes on the token to the next processor. A new *RCU epoch* starts with completion of a full round trip of the token. By then it is guaranteed that all processors have passed a quiescent state; thus the effects of an operation of the previous epoch are complete and globally visible.

Operating systems have well-known code paths, where it is structurally guaranteed that the current thread does not hold locks and thus are in a quiescent state. Examples of these code paths include when a thread exits from kernel to application level, as well as when the processor enters the idle loop, or on context switches.

Read-copy update is used in a variety of scenarios in operating systems, such as deferred deletion of list members or module existence locks for kernel-loadable modules. In this thesis I apply the core idea of RCU to a novel domain: *kernel lock primitives*.

3.3 Partitioning

Multiprocessor partitioning divides a large multiprocessor system into a set of independent, smaller multiprocessor subsystems. Each subset acts relatively autonomously and independent of the others. The strict independence between partitions eliminates interference, information leakage as well as restricts the level of parallelism within each partition. The scalability requirements for an operating system in a partition are therefore limited by the physical resources (i.e., processors) allocated to it.

Partitioning shares a subset of goals with scalable microkernel-based systems. The hard boundaries eliminate the overhead induced by a high degree of parallelism, such as cache and TLB coherency, and also restrict the degree of parallelism and thus the possibility for contention. Hence, operating systems that run *within* the partition, can act as if they would run on a smaller overall system, even though the hardware may have significantly more processors. The OS can reduce overhead for precautions for processors that are not supported and use the best synchronization scheme for the partition's specific resource configuration.

3.3.1 Hardware Partitioning

Hardware partitioning is a well-known technique from mainframe class computers and was recently introduced on enterprise-class UNIX systems. Hardware resources, including processors, memory and I/O devices, are allocated to individual partitions. The partitions are strictly isolated from each other via specialized hardware support.

Brown Associates [32] compares the Sun E10000 and Unisys ES7000 systems that both support hardware partitioning. The Sun E10000 [102] supports up to sixteen domains that achieve complete isolation while sharing the interconnect. Registers on the system boards define the domain per board. All components controlled by the system board become part of the domain. Failures within a domain do not affect other domains, except for failure of system-wide components.

The Unisys ES7000 [112] supports up to eight partitions and the quantity of processors, memory, and I/O devices in a partition can be freely configured. Resources between partitions can be adjusted dynamically. The ES7000 memory sys-

tem supports private memory and memory to be shared among partitions. Similar hardware partitioning is supported by IBM Sequent, and HP (nPar).

The limiting factors of hardware partitioning are the *relatively static resource allocation* and the *strict partition boundaries*. Resources that are allocated to one partition cannot be easily shared by another. In particular it is not possible to share processor resources between partitions, in case a processor falls idle while another system is overloaded.

An important feature of partitioning is fault containment. The two possible failure types are hardware failures and software failures. While microkernel-based systems naturally achieve containment of software failures of applications via address space protection, hardware fault containment requires specific support in the microkernel (hardware fault containment is not addressed in this work).

3.3.2 Software Partitioning and Virtual Machines

Similar to the hardware based partitioning approach, software partitioning multiplexes the hardware resources between different operating system instances. Each OS instance has a subset of the overall system resources.

Software partitioning is often used in the context of virtual machines (VM) [43], where multiple concurrent operating systems compete for the system resources. The virtual machines are under control of a privileged virtual machine monitor (VMM) that serves as the controlling entity and resource manager. VMs and microkernel-based systems are very similar in respect of the isolation goals. They differ insofar that microkernel-based systems provide a set of abstractions that enable fine-granular resource isolation and recursive resource control for all system services, whereas most VMMs solely target for monolithic coarse-grain hardware multiplexing [15]. As our group has shown, a microkernel can serve as the core platform for virtualization [69, 111], while the opposite is often not the case because of the lack of high-performance communication primitives.

Besides our work on scalable virtual machines [111], the other most prominent research addressing VMs for multiprocessor systems is Cellular Disco [47].[2] Cellular Disco aims to provide the advantages of hardware partitioning and scalable operating systems while avoiding the scalability bottlenecks. Cellular Disco uses a small, privileged VMM that multiplexes resources of different virtual machines. The system is divided into cells that act independently and interact via an in-kernel RPC mechanism. A primary goal of the messaging system is to preserve independence for fault containment and to ensure survival of cells in case of remote failures.

While Cellular Disco shares some goals with my thesis, it differs in the way these goals are achieved. Cellular Disco specifically targets VMs and provides the core services, such as scheduling, page allocation, and device drivers *within* the privileged VMM. Thus, the VMM follows a traditional monolithic operating

[2] That does not include commercial systems, such as IBM's OS 390.

system design paradigm, with in-kernel policies.

A key design decision of Cellular Disco is to assume that the code is correct. The assumption is warranted by the fact that the size of the virtual machine monitor is small. The authors report a prototype kernel of 50K lines of code (compared to 13K for L4Ka::Pistachio), however, it is *co-located* with an SGI IRIX 6.4 OS for device accesses. Despite the inherent dependence on IRIX for correctness, such co-location also has implications on scalability. The scalability of an important aspect of the system — hardware device accesses — becomes dependent on the overall scalability of IRIX. Also, the work does not specifically address scalability of multiprocessor systems but has a stronger focus on general problems of VMs, such as eliminating double paging and memory balancing policies.

3.4 Microkernel-based Systems

3.4.1 The Microkernel Argument

A common method for structuring operating systems is a *monolithic system*, which integrates all OS functionality in a single system image (e.g. UNIX [93] and VMS [44]). The main advantage of such a monolithic structure is good performance and global visibility of system state. Monolithic systems, however, have a number of disadvantages. Since all system services reside within the monolithic system image, all subsystems share a single protection space and may interfere. Faults within one subsystem can propagate to others with potentially fatal consequences for the system.

In order to extend such systems with new functionality such as different scheduling policies, file systems or device drivers, the extension has to be *integrated* with the monolithic kernel. The extension is code that is injected in the kernel and runs with full privileges. Thus, the injected code becomes globally trusted, however may neither be trustworthy nor error free. A variety of approaches address trustworthiness, such as simple code signing [33,83], sandboxing [119], interpreted languages [84], type-safe languages [18], domain-specific languages [40], and proof-carrying code [85, 86].

In contrast to monolithic designs, in microkernel-based systems the operating system functionality is implemented as *user-level servers*. These servers run in separate address spaces and communicate using inter-process communication (IPC) built into the microkernel. Extensibility is achieved by replacing the system servers, similar to normal applications.

Microkernels isolate operating system components via address spaces from each other and also provide them with separate resources. Both properties enable multiple OS services with different resource policies to co-exist on one system. These include systems with different security requirements [90] or systems with real-time and non-real-time demands [53, 97].

The second important property is the size and complexity of microkernels. Microkernels are tiny and thus can be used as a secure platform for custom, applica-

tion specific operating systems [62, 75].

First-Generation Microkernels

First-generation microkernels failed to fulfill the promise for better structured and more flexible systems. The most prominent kernel is Mach [2] and it provides a good example of the problems of first-generation microkernels. Mach was designed to exploit modern hardware features such as multiprocessors, and enable new application domains. At the same time, Mach was supposed to provide backward compatibility to existing operating systems such as UNIX [93]. As a result, Mach was constructed by refactoring a UNIX kernel and introducing new kernel interfaces. The UNIX functionality was moved to application servers.

Mach lead to a number of important innovations, such as user-level pagers [123], with the result of a tremendous growth of the API surface area and an increase of kernel complexity and size. The primary problem of Mach, however, was its opulent and extremely slow IPC mechanism that resulted in severe performance problems for systems constructed on Mach.

Second Generation Microkernels

Second generation microkernels addressed the above problems with radical new designs. Liedtke [71] identified poor structuring and high cache footprint of Mach's IPC primitive as the main performance problems. By careful design of the communication mechanism, second generation microkernels such as L4 [71], Exokernel[3] [34, 62], and EROS [98] exhibited the envisaged properties with superior performance to other extension methods.

The fundamental design principles for most second generation microkernels are:

- The kernel provides a *minimal set of abstractions*: *threads* and *address spaces*.[4]
- The kernel provides a *minimal set of mechanisms* in order to multiplex hardware in a safe and secure manner. The key primitive is inter-process communication which enables extensibility of the system via interacting user-level servers [73].
- The kernel should be *policy free*. The policies should be provided by the system servers using the kernel mechanisms.

[3] Exokernels differ from microkernels insofar that exokernels avoid high-level abstractions but directly multiplex the hardware. I still discuss them with microkernels because many goals are very similar.

[4] More general, each microkernel needs at least one abstraction for execution (or an activity) and one abstraction for a protection domain to permanently bind resources.

Based on these design principles a number of microkernels were developed that yielded excellent performance for both system software and user applications. Härtig et al. [54] showed that L4 not only has excellent IPC performance but also incurs a minimal overhead when running a monolithic operating system (Linux) on top of the microkernel. The reported overhead for macrobenchmarks was between 5 to 10 percent compared to monolithic Linux. Still, microbenchmarks showed overheads of a factor of three for short-running system calls. I and colleagues [110] further reduced the microkernel-induced communication overhead from a factor of three to about thirty percent.

3.4.2 Multiprocessor Microkernels

One aspect of previous research in the area of microkernels is multiprocessor systems. The reduced complexity of microkernels as compared to monolithic systems makes them a feasible system architecture.

The Hydra kernel [122] was designed with the goal of separating mechanisms and policy and thereby provides an exploration testbed for multiprocessor systems.

Mach targets large-scale multiprocessor systems and provides lightweight threading in order to utilize the parallelism offered by the hardware [20]. Being a first-generation microkernel, Mach employs a traditional scalability approach comparable to many monolithic multiprocessor OS's. The in-kernel subsystems were optimized on a case-by-case basis [64] by first identifying and then removing scalability bottlenecks. However, Mach still exhibited its poor IPC performance.

The Raven microkernel [94] specifically targets multiprocessor systems with a focus on minimizing the overhead for kernel invocations. Raven provides user-level scheduling with a kernel-based callback (similar to scheduler activations [5]), user-level interrupt handling, user-level interprocess communication, and user-level synchronization primitives. While moving significant system parts to application level, Raven uses a number of global data structures, thus limiting its scalability. For example, the memory pool is protected by a global lock and accesses to task objects are serialized.

Chen [27] extended the uniprocessor Exokernel system to multiprocessor systems. The fundamental idea of Exokernel is to expose the hardware properties directly to application level thus avoiding the overhead introduced by high-level OS abstractions. The OS itself is implemented as a library at application level. The multiprocessor Exokernel uses a variety of alternative synchronization strategies. Similar to my approach described in this thesis, Exokernel localizes kernel resources and uses in-kernel messaging to manipulate remote resources. However, the choice for synchronization primitives is ad-hoc and static. The specific kernel implementation dictates the synchronization granularity and may result in high overheads. This is, for example, reflected in the memory subsystem, where the reported cost for memory mappings is three times higher than on the uniprocessor version of the kernel. Furthermore, the overhead for the kernel's IPC primitive (even for uniprocessor systems) is almost an order of magnitude more expensive

than L4.

The most prominent work in the area of scalable microkernel systems is Tornado [41] and its successor K42 [8]. K42 is a microkernel-based system that employs many of the previously described techniques for scalability, including read-copy update, clustered objects, and lock-free synchronization. K42 scales well to a large number of processors, provides a Linux-compatible API, and reuses Linux components in the kernel [9]. K42 differs from my approach insofar, as K42 blurs the line between kernel and application level and allows the kernel to be extended in order to overcome performance bottlenecks. For example, K42 allows for co-location of device drivers if performance requires. K42's underlying microkernel provides access to kernel-internal data structures via a wide kernel-user interface (e.g., for scheduling information). K42 also achieves adaptability of synchronization mechanisms, however, via runtime code replacement and with the limitations of the fault-based scheme described in Section 3.1.2.

L4

Besides the work presented in this thesis, there are at least two other L4 microkernel variants that support multiprocessor systems. Liedtke [70] briefly discusses the implications of multiprocessor support and argues that locking will have a significant performance impact on all critical system operations including IPC. Potts et al. [91] realized a multiprocessor version of L4 on the Alpha architecture with the primary focus on the performance of the local-case IPC. All inter-processor operations use message-based synchronization schemes. The kernel, however, does not support kernel memory pool management and also lacks a TLB coherency scheme.

Hohmuth [58] describes Fiasco, an L4 kernel that focuses on real-time properties via non-blocking synchronization. Fiasco uses a locking-with-helping scheme in order to minimize thread blocking time in real-time systems. If thread *A* tries to acquire a lock that is held by a preempted thread *B*, *A* donates its time to *B*. After the lock is released, *B* immediately switches back to *A*. Fiasco's helping scheme is also used across processor boundaries. The approach, however, is questionable for larger multiprocessor systems for the following reasons:

Helping may lead to temporary migration of threads between processors. The overhead induced by the required cache migrations (in particular the active working set of the thread's kernel stack) is significant. Even small kernel operations require multiple cache lines to be migrated. The probability for a hot cache working set of a preempted thread in L4 is extremely high due to the very short execution time of kernel operations. Furthermore, the dynamic migration effectively requires all kernel objects to be protected with atomic lock operations. Hohmuth reports an increase in cost for the *kernel thread lock* on multiprocessor systems of 209 percent for a Pentium 4. The reported overhead for the lock-with-helping primitive (that is acquired on the critical path) is *four times higher* than a complete IPC operation in L4Ka::Pistachio. The work does not discuss scalability aspects, such as kernel memory management, scheduling, TLB coherency, or the implications of helping

on overall scalability.

Hohmuth argues that a helping scheme is more general, because it is not limited to microkernels, where operations are in most cases very short. My work *specifically targets* microkernels and the trade-off of generality vs. performance clearly favors performance. My goal is to achieve maximal performance and scalability without requiring extensibility of the kernel. The base-line is thereby the uniprocessor version of L4Ka::Pistachio with its unrivaled IPC performance on almost all existing hardware platforms.

Chapter 4

Microkernel Performance Adaptation

In this chapter I present an approach for controlling and adjusting performance- and scalability-relevant parameters of multiprocessor systems. My approach is based on efficient tracking of parallelism and dynamic adaptation of multiprocessor synchronization mechanisms (i.e., locks). My goal was to develop a set of general mechanisms that lead to a system construction methodology with excellent scalability properties and low overhead while preserving strict orthogonality and isolation between independent subsystems.

The mechanisms and methodologies target microkernels, component-based systems, and virtual machines on small and large scale multiprocessor systems with up to about one hundred processors. Some mechanisms are usable beyond this specific domain. The dynamic locking scheme I developed can serve as a drop-in replacement for standard spin locks and clustered locks. The processor tracking scheme is a drop-in replacement for processor sets as widely used in scalable UNIX variants.

This chapter is organized as follows: Section 4.1 defines the problem that my approach intends to solve, states the general scalability and performance goals, and narrows the target hardware architecture. Afterward, I derive a set of requirements to achieve the performance goals, which I will discuss in the remaining sections of this chapter. In Section 4.2, I describe a mechanism to efficiently track and restrict resource usage to a subset of processors. Section 4.3 then compares the overhead of kernel synchronization mechanisms and I propose a scheme that allows for safe runtime adaptation depending on the most efficient algorithm. Section 4.4 describes a TLB coherency tracking scheme that decouples permission updates from TLB shoot-downs thereby reducing TLB coherency overhead and minimizing inter-processor dependencies. Finally, Section 4.5 describes a generic event logging mechanism that enables user-level components to efficiently monitor and evaluate events across component boundaries.

4.1 Overview

The design guidelines for scalable operating systems, as defined by Unrau [113] (preserving parallelism, bounded overhead, processor locality), leave performance and overhead aspects of multiprocessor algorithms unaddressed. As argued before, by strictly following these three guidelines, a system is still considered scalable even if the overhead is multiple orders of magnitude higher than the achievable minimum.

Minimal overhead and maximum performance for a particular algorithm depends on a variety of parameters, most importantly the degree of parallelism and concurrency and the (memory) interconnect between the involved processors. The decision for one or the other alternative depends on the specifics of the system (both, software and hardware), access pattern, access frequency and so on. It is possible to optimize on a case-by-case basis when incorporating *semantic knowledge* on resource usage. In operating systems, such semantic knowledge is available for many high-level abstractions, such as the degree of sharing of a file, the number of threads in an address space, physical memory that backs a device and therefore is reallocated infrequently, and many others.

In microkernel systems such high-level system abstractions are implemented as user-level services. Hence, detailed semantic knowledge is not available to the microkernel itself. Instead it operates on the bare hardware resources: memory, time, and the processor resources (e.g., caches and TLB). These resources are all treated identically, independent of the specific usage case. The lack of the detailed high-level information either leads to an overly conservative design that favors scalability over performance, or a well-performing but less scalable system. Introducing more semantic knowledge via high-level abstractions, however, defeats the minimality principles of microkernels.

From that starting point I define the following primary design goals:

- The overhead for kernel operations in a multiprocessor has to reflect the degree of parallelism and concurrency. Semantic knowledge about resource usage that is available at the *application level* needs to be considered at the *kernel level*.

- The microkernel abstractions and mechanisms should remain minimal with a strict focus on orthogonality and independence. The microkernel itself should not be extended by multiprocessor-specific *policies* in order to achieve better scalability or less overhead. Instead, the kernel must provide mechanims such that applications can adjust and fine tune the kernel's behavior.

- All optimizations and adaptations must be safe and enforce protection and isolation.

Furthermore, I define a set of secondary design goals:

- A scalable OS design has to honor Unrau's design principles: preserve parallelism, bounded overhead, and memory locality in NUMA systems. That requirement is for the microkernel and also for OS components at application level.
- The design must be efficient on a variety of multiprocessor systems, from small-scale shared memory to large-scale NUMA architectures. The supported number of processors must be independent of architectural specifics, such as the number of bits per register.
- The architecture should target contemporary hardware with high memory access latencies, high cache-coherency overhead for NUMA configurations, and potentially long IPI latencies. My specific target hardware architecture is IA-32.
- The performance baseline is the uniprocessor kernel. Operations that are restricted to a single processor should have the same overhead as if executed on a single processor system.

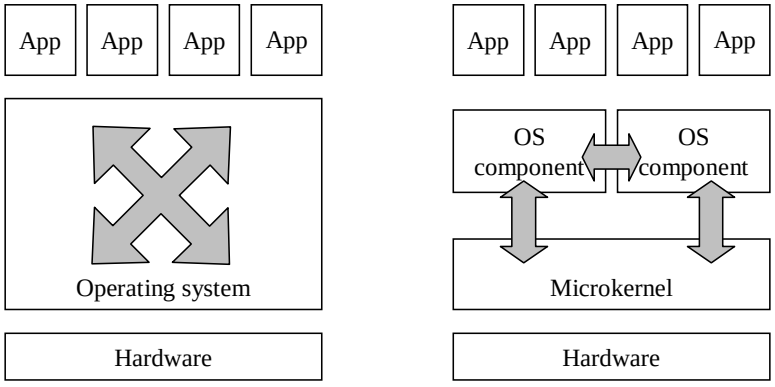
Approach

The separation of operating system functionality into isolated components restricts information availability and information flow and thus limits microkernel-based systems in their optimization potential. Figure 4.1 gives a schematic comparison of information flow in a monolithic OS versus a microkernel-based system. To achieve the same flexibility and optimization potential for the microkernel-based system requires (*i*) an efficient information flow between the kernel and components, and (*ii*) mechanisms to control and adjust a system's behavior.

While these are general requirements, I will now narrow them to the specific problem domain of multiprocessor systems and scalability. In order to achieve optimal performance, the kernel has to adapt alternative synchronization strategies and synchronization granularity. They depend on the *degree of parallelism*, the *frequency of operations*, and the *granularity* at which individual objects are modified. The information has to be provided to the kernel by applications and OS components and furthermore need to reflect in the kernel's algorithms and data structures. By breaking high-level semantic information on resources down to a set of generic attributes, it is possible to provide those to the kernel in a unified manner.

I developed the following four mechanisms to counteract the limitations that are inherent to the microkernel's system structure:

Tracking of parallelism. Applications can explicitly specify the expected parallelism for a kernel object and the kernel keeps track of it. Based on that information, the kernel can make an informed decision on the synchronization



(a) Monolithic system (b) Microkernel system

Figure 4.1: Schematic comparison of information flow in (a) monolithic systems and (b) microkernel-based systems. In a monolithic system all information is combined in a single system image. A microkernel-based system requires mechanisms to control and adapt the behavior of components and mechanisms for efficient control flow.

strategy. Explicit specification overcomes the lack of semantic information about parallelism of kernel resources.

I developed a fixed-size data structure that enables efficient tracking of parallelism and locality for every kernel object and forms the basis for adaptive locking.

Adaptive locking. The usage pattern of resources may change over time. Applications need to be able to readjust the system's behavior and need to adapt the synchronization scheme, such as lock granularity and the synchronization primitive.

TLB coherency tracking. Memory is a primary resource that is managed by the microkernel and is the basis for isolation of components. Memory permissions are stored in page tables and cached in the processor's TLB. After revocation of memory permissions, the kernel has to enforce TLB coherency, which is an expensive operation. By combining multiple TLB updates into one when manipulating a set of page table entries, the shoot-down overhead can be reduced. However, the combined update has similar negative implications on parallelism as critical section fusing. I developed a tracking algorithm that decouples permission updates and TLB shoot-downs. The algorithm uses a TLB version that allows for combining multiple TLB shoot-downs in one. The inter-dependencies between processors is eliminated by bypassing outstanding shoot-downs.

Event tracing. Feedback-based policies incorporate runtime information that is

derived from monitoring system behavior. In multiprocessor systems, many resource allocation policies — including thread balancing and memory pooling — are based on monitoring the system's behavior and predicting the future. In a component-based system, the information is distributed and thus unavailable to schedulers. Event tracing provides an efficient mechanism to monitor and transfer relevant data between the kernel and components but also across components.

Having relevant kernel information at hand, avoids the necessity of introducing multiprocessor kernel policies and leaves such decisions to applications. A prerequisite of such an approach is minimal overhead for data collection, transport, and evaluation.

In the following sections I describe these four mechanisms in detail.

4.2 Tracking Parallelism

Multiprocessor optimizations depend on specific knowledge of the *degree of parallelism* on data structures and the *locality* of operations. Based on both properties, the operating system and applications can select the most suitable (e.g., best performing) algorithm for a specific task. In monolithic systems, in many cases it is either possible to derive or imply the degree of parallelism from the system structure. For example, it is unlikely that the data structure representing a file, which is opened by a single-threaded application, will experience high contention by many processors; the most likely processor that accesses the data structure is the home processor of the thread. In this example, the relevant information is encoded indirectly by (*i*) the locality of the thread(s) and (*ii*) the number of threads that may manipulate the file.

If such indirect knowledge on parallelism and locality is unavailable it can alternatively be explicitly expressed with a *bitmap*. Each bit in the bitmap denotes one processor in the system. The degree of parallelism is encoded by the number of bits set and locality is expressed by the specific bit position. This encoding scheme is efficient and accurate, however, *it does not scale*. The size of the bitmap grows linearly with the number of processors in the system. For kernel objects (e.g., the previously mentioned file structure) that contain a processor bitmap, the memory footprint and cache footprint depends on the number of processors in the system. Also the runtime overhead for deriving the number of bits set depends on the size of the bitmap and thus the total number of processors. Both properties (i.e., unbounded size and unbounded runtime) violate Unrau's scalability requirement of bounded overhead.

Tracking processor affinity and processor permissions requires an efficient, fixed size encoding. Processor sets (as used by many UNIX kernels) solve the space problem by a level of indirection; the bitmap is replaced by a pointer to the bitmap. However, such an indirection scheme not only increases the runtime over-

head but has also the disadvantage that it is non-trivial to derive whether or not the bitmap is still referenced.

I developed an encoding scheme that has both desired properties, it has a fixed size and is independent of the number of processors. Similar to floating point encoding, I trade accuracy for space. The trade-off limits the freedom to combine arbitrary processors in the mask, which is no limiting factor in the common cases of resource allocations for the following reasons. Non-uniform memory increases the cost for remote accesses. Any *well designed* system should therefore decrease the frequency of accesses to remote resources with increasing cost. The most common and relevant approaches for reducing the remote access frequency are (*i*) to explicitly favor local over remote resources (and reallocate resources on migration), (*ii*) minimize resources that are shared across many processors (e.g., by replication), and (*iii*) co-locate tasks that have a high degree of sharing on neighboring processors.

In order to minimize overheads, a well-designed operating system for multiprocessor architectures will therefore try to minimize the degree of sharing of physical resources with increasing access costs. One can conclude that such systems will preferably limit resource sharing to nearby processors, that is processors with low access latencies to the same memory.

4.2.1 Processor Clusters

The specific resource access costs depend on the memory interconnect and its physical limitations. Shared bus interconnect do not scale due to bandwidth limitations, while point-to-point connections do not scale because of the quadratic number of interconnects between processors. The commonly used topologies, such as hypercubes [55, 56] and fat trees [67], trade performance against hardware complexity and have only very few neighboring processors. The specific number depends on a variety of hardware factors. For example, most NUMA systems combine a few processors via a shared memory bus which then connects via a memory router to other processors. In multi-core and multi-threaded systems, processors share one memory subsystem and in some cases processors even share the same caches.

Based on the limits of the memory subsystems one can conclude that in *any well-designed* multiprocessor system:

1. most resources will be accessed by only a very small subset of processors,
2. the processors that share the resources will be located relatively close to each other.

Based on that observation, I developed an encoding scheme that avoids both scalability problems stated before. I use an encoding scheme that is specifically tailored towards the common scenario of neighboring processors. First, I group processors in clusters of neighboring processors. The clusters are constructed by considering the access latencies to shared resources. Each processor has its unique

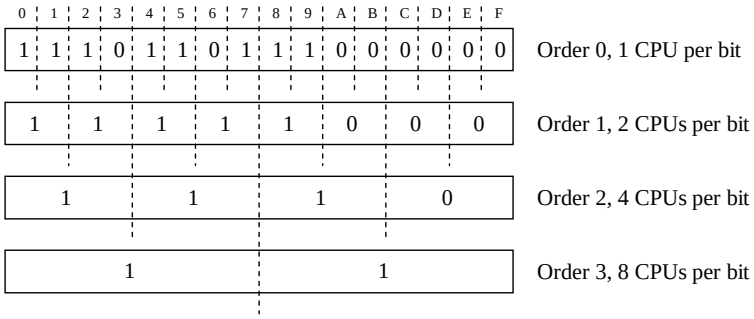


Figure 4.2: Cluster-mask encoding for different cluster orders. The sample values show the reduction of accuracy with increasing order.

numerical ID in the system. I assign the IDs in a way that neighboring processors have neighboring IDs. The organization of the memory interconnect makes the ordering process straight forward: first sort processors of the same core, then same node, and so on.

Similar to the normal bitmap encoding, where one processor is expressed by one bit, I encode *one cluster* per bit. The scheme can be applied recursively resulting in a cluster hierarchy. Multiple clusters are then covered by a single bit. The number of processors per bit is encoded in the *cluster order* field as a power of two.

Obviously, when grouping state about multiple processors into a single bit, the encoding looses on accuracy. Figure 4.2 shows an example encoding of a processor set with different cluster orders. The example has alternative encodings for a bitmap covering 16 processors. Starting with a cluster order of 0, each processor has an individually assigned bit. The encoding is accurate and requires as many bits as processors. For the cluster order of 1, 2 processors are encoded per bit. The encoding requires half the number of bits but also reduces accuracy by 50 percent. While in the initial encoding the bitmap selects 8 processors, the order 1 encoding is less accurate and selects 10 processors (respectively, 12 processors for order 2 and 16 processors for order 3).

4.2.2 Processor Cluster Mask

Processor clustering reduces the number of bits required to cover a large number of processors at the cost of accuracy. As argued before, for performance reasons many operations on resources are restricted to only a subset of the system's processors. The specific processors depend on the resource access costs. The described assignment of processor IDs for physically neighboring processors favorably encodes them in neighboring bits in the bitmap.

Instead of encoding *all* processors in the bitmap, I restrict the bitmap to cover a subset of the processors only. The subset of processor IDs is encoded by a start ID (or offset) and the cluster order. The number of bits in the bitmap give the

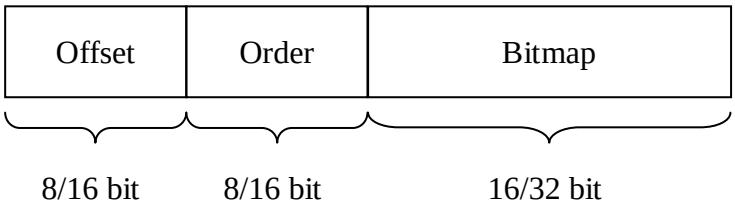


Figure 4.3: Cluster mask encoding for 32 and 64 bit systems (left for 32 bit, right for 64 bit). The structure size is identical to natural word width of the processor.

upper bound of the subset. For example, a bitmap with 16 bits and an order of 1 covers a total of $16 \cdot 2^1 = 32$ *neighboring* processors. In comparison, the same-size bitmap but with an order of 5 covers $16 \cdot 2^5 = 512$ neighboring processors. The start processor ID enables to *offset* the covered range of the bitmap. The processor IDs that are covered by a bitmap with b bits are in the interval $(2^{\text{order}} \cdot \text{offset}, 2^{\text{order}} \cdot (\text{offset} + b)]$.[1]

The cluster bitmap can encode very large numbers of processors in a fixed-size data structure. For the binary representation I defined the following set of requirements:

1. The data structure must have a size that is identical to the processor's natural register width, such that it can be efficiently stored, efficiently passed around (i.e., from application to kernel in a register), and used with atomic operations (such as compare-exchange).

2. Testing for a processor must have very low overhead, because such operations are in many cases performance critical.

3. The data structure has to provide identical accuracy for *all* processors in the system (i.e., there should be no penalty for processors with a particular ID).

4. The runtime overhead of operations on the data structure (e.g., lookup, test, or merge) must be bounded independent of the number of processors in the system.

These four goals are fulfilled by the data structure depicted in Figure 4.3. It contains three parts, (*i*) a bitmap for encoding individual processors, (*ii*) a field that denotes the processor order, and (*iii*) an offset field that specifies the starting processor ID of the bitmap.

A processor p in a system with a maximum of $p_{\max} = 2^k$ $(k \in \mathbb{N}_0)$ processors is selected by the cluster mask $m = (\text{bitmap}, \text{order}, \text{offset})$, if $f(m,p) > 0$ with

$$f(m,p) \equiv \text{bitmap} \bmod 2^{((\lfloor p+p_{max} \rfloor / 2^{\text{order}} \rfloor - \text{offset}) \bmod \lfloor p_{max}/2^{\text{order}} \rfloor}.$$

[1] Note that the interval ignores wrap-arounds.

The equation expresses a bit test using the cluster encoding of Figure 4.2. It can be efficiently implemented via simple rotate and logical bit operations.

Furthermore, I define a set of operations for the cluster mask, most importantly the *merge* operation for two cluster masks. Merge is similar to a logical OR operation insofar that it combines the bits of two bitmaps into a new bitmap. Additionally, it adjusts the order and offset such that all previously specified processors are still selected by the new mask. The cluster mask m' that is merged from m_1 and m_2 must hold the condition $\forall i \in (0,1,\ldots,n-1) : f(m_1,i) + f(m_2,i) > 0 \Rightarrow f(m',i) > 0$. For cluster masks that have identical orders and offsets, the merge operation is a simple logical OR operation of both bitmaps.

The number of processors specified by the cluster mask is the number of set bits in the bitmap multiplied by 2^{order}. Furthermore, the order field serves as an indicator for optimizing the locking strategy. A higher value for the order encodes more processors (thus a higher probability for contention) and also potential overhead and unfairness in the memory subsystem. The next section covers this aspect in more detail.

4.3 Kernel Synchronization

The kernel uses synchronization to *ensure consistency* of data in the kernel and to *order operations*. The two primary synchronization schemes available in shared-memory multiprocessors are memory-based locks (and atomic operations) and in-kernel messages. Both schemes have advantages and disadvantages and there is no single optimal solution.

I propose to provide multiple alternative synchronization schemes for the same data structures. Applications can then choose the optimal strategy based on their specific local knowledge on access and usage patterns. The degrees of freedom of the locking strategy is the *lock primitive* (i.e., memory- or message-based) and the locking *granularity* for divisible kernel objects. In case of no parallelism, the lock primitive is unnecessary and only induces runtime overhead and additional cache footprint and could be disabled altogether.

The cluster mask described in 4.2.2 enables the kernel to efficiently store and evaluate the degree of parallelism on a per kernel-object basis. By tagging each kernel object with such a mask, applications have detailed control over the used synchronization primitives for each individual object. Processors that are specified in the cluster mask can manipulate kernel objects directly using memory-based synchronization; all other processors need to use message-based synchronization. The cluster mask provides the required information for an informed decision. It also *enforces* that only those processors specified in the mask have direct memory access to the object. For a mask that specifies a single processor the lock primitive can be safely *disabled*.

In this section I first discuss the performance trade-offs of the alternative synchronization schemes. The better-performing scheme thereby depends on the spe-

cific access pattern. The choice for one of three synchronization schemes — coarse-grain locking, fine-grain locking and kernel messages — has to be made on a case-by-case basis and may change over time. A static selection becomes suboptimal when locking pattern and parallelism significantly changes over time.

I propose a synchronization scheme that eliminates this deficiency by incorporating the additional knowledge provided via the processor cluster mask. When the cluster mask changes, the kernel *adjusts* the synchronization primitive between normal locks and message-based primitives. A problem of this approach is that updates to the mask are not propagated instantaneously to all relevant processors. Thus, during a transition period some processors may use the previous, while others the new synchronization primitive.

However, incorrect (or inconsistent) synchronization could lead to data corruption in the kernel and erroneous behavior. I describe a new locking primitive — *dynamic locks* — that can be safely adjusted at runtime and guarantees correctness during the transition period from one locking scheme to another. Dynamic locks can be enabled and disabled at runtime and allow the kernel to dynamically eliminate locking overhead for unshared resources. Built upon the base primitive of dynamic locks I develop a scheme that cascades multiple dynamic locks in order to dynamically and safely adjust the *lock granularity* addressing the cost–concurrency trade-off.

4.3.1 Trade-offs

The performance trade-off for in-kernel synchronization is based on three parameters: (*i*) ratio of local vs. remote access, (*ii*) the overhead of the lock primitive, and (*iii*) the degree of concurrency. Based on these parameters, one or the other synchronization method is more efficient.

Message-based synchronization. In message-based synchronization a datum has an associated home processor. Only the home processor can manipulate the datum, thereby achieving mutual exclusion and strict ordering. Message-based synchronization incurs no overhead for manipulations on the home processor, however, a significantly higher overhead for accesses from remote processors. Hence, the overall cost depends on the fraction f_{remote} constituting remote operations of the overall operations, with the total synchronization cost $t_{\text{total}} = f_{\text{remote}} \cdot t_{\text{message}}$.

The overhead for message-based synchronization can be divided into the cost for message setup and signaling cost, the delivery latency of the message, and the overhead for processing. Signaling cost may require an interprocessor interrupt that induces cost on the sender side and also on the receiver side (for the interruption itself and interrupt acknowledgment on the hardware).

After sending a message, the sender can either actively wait for completion or block and invoke the scheduler. In the latter case the latency of the overall

operation increases, but the initiator of the message does not busy-wait until the message gets delivered to the remote processor. Busy-waiting adds the message delivery latency to the overall cost of the operation. When invoking the scheduler, the processor is freed for other tasks that can be executed while the in-kernel message is in transit. However, the context switch may induce a performance penalty later on for replacements of the preempted task's active cache working set (memory and TLB).

Lock-based synchronization. Locks provide mutual exclusion based on memory ordering that is enforced in the memory subsystem. The overall overhead for locks can be divided into the cost for the lock operation (instruction execution and cache footprint) t_{lock}, and the wait time t_{wait} when the lock is already taken. The average lock wait time depends on the number of competing processors p and the average lock holding time t_{hold} and is $t_{\text{wait}} = t_{\text{hold}} \cdot (p-1)/2$. The cache coherency updates additionally induce indirect costs in the memory subsystem.

More sophisticated synchronization primitives, such as semaphores, can release the processor to other runnable tasks. Such primitives, however, are in most cases too heavyweight for the short critical section of microkernels and the overhead outweighs the potential benefits.

The choice between one or the other synchronization scheme depends on the specifics of the workload. Following, I compare the costs and derive a decision metric. The metric is based on the previous simplistic cost analysis. Hence, it leaves many aspects of algorithms and hardware unaddressed, however, it is sufficient to reflect the general performance and overhead tendencies.

Message-based synchronization favors local operations over remote. It eliminates the locking overhead in the local case. Hence, message-based synchronization has a lower overhead if the overall cost for locks (for the local and the remote case) is higher than the cost for sending and processing the message including all additional costs (e.g., interrupt handling, interrupting processing on the remote processor, etc.). Message-based synchronization is therefore preferable for infrequent remote operations or when multiple remote operations can be combined in a single remote request for minimizing the startup overhead.

Lock-based synchronization yields better overall performance compared to message-based synchronization, if the cost for locks on processor-local data is less than the message overhead from remote processors.

Another performance aspect is the lock granularity for *divisible* objects. Depending on the degree of concurrency, the lock overhead, the number of concurrently manipulated parts, and the overall lock holding time, either coarse or fine grain locks may result in better overall performance.

Instruction	Opteron 1.6GHz	P3, 800MHz	P4 Prescott 2.8GHz
xchg	16 cycles	19 cycles	121 cycles
lock cmpxchg	18 cycles	26 cycles	139 cycles
lock decb	18 cycles	22 cycles	131 cycles

Table 4.1: Cost for atomic instructions for different IA-32 processors

Let's consider an object that can be divided into n independent parts (e.g., a hash table). The runtime overhead for a lock acquisition is t_{lock} and the lock hold time is t_{hold}. The average wait time for p competing processors is $t_{\text{wait}} = (p-1)/2 \cdot t_{\text{hold}}$. The total synchronization overhead t_{total} to modify m out of n parts with p processors evenly competing for all parts is $t_{\text{total}} = t_{\text{wait}}/n + m \cdot t_{\text{lock}} = (p-1)/(2n) \cdot t_{\text{hold}} + m \cdot t_{\text{lock}}$

The following two common cases are of particular relevance:

- *No or low concurrency:* For $p = 1$ the overall cost is $t_{\text{total}} = m \cdot t_{\text{lock}}$. The synchronization overhead depends on the number of modified parts m. m can be reduced by decreasing the granularity (i.e., by using coarse-grain locks).
- *High concurrency:* For $p \gg 1$ the overall cost is dominated by the wait time $t_{\text{total}} = (p-1)/(2n) \cdot t_{\text{hold}}$. It can be reduced via fine granular locking (i.e., increasing n) or by reducing the lock holding time.

Please note that this analysis is very simplistic. In particular it does not include cache effects, such as multiple acquisitions of the same lock, or the additional overhead induced by cache coherency overhead in NUMA systems. It also assumes an equal distribution of lock acquisitions between all locks.

For today's architectures the cost of t_{lock} is increasingly high and for short critical sections even more expensive than the actual operation that is protected by the lock. Table 4.1 compares the *minimal* overhead for atomic instructions on a variety of IA-32 processors.

The fundamental idea is to use the available application-specific knowledge on the degree of concurrency and access pattern (local or remote) and adapt the kernel's synchronization mechanisms. The adjustable parameters are the overhead of the lock primitive t_{lock} and the lock granularity n. The lock primitive can be adjusted in two ways. First, it can be eliminated by switching from lock-based synchronization to message based synchronization. Message-based synchronization has a lower runtime overhead for $f_{\text{remote}} \cdot t_{\text{message}} < t_{\text{lock}}$.[2] Alternatively, the locking scheme can be adapted, for example by switching from spin locks to MCS locks as proposed by Lim [76].

[2]Note that the inequality does not address the implication of higher latencies on overall performance.

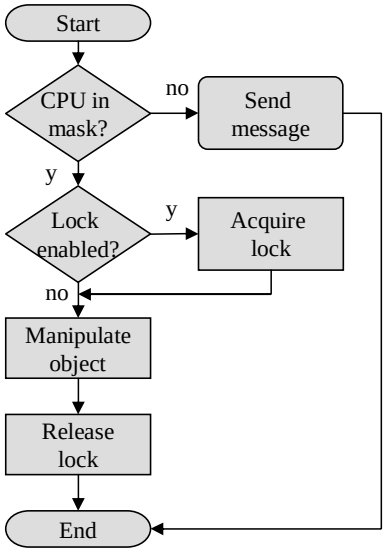


Figure 4.4: Flow chart of dynamic kernel synchronization. If the current processor is not specified in the cluster mask, the algorithm can either use an in-kernel messaging scheme or fail (not shown).

Adjustment of the synchronization scheme is based on explicit application feedback. As opposed to applications, the kernel cannot trust any application claims of non-existing parallelism but needs to enforce correctness. On every resource access the kernel explicitly validates correctness. Figure 4.4 depicts the general control flow for adaptive in-kernel synchronization. Kernel objects are tagged with a cluster mask. If a processor is not specified in the mask, the object can not be manipulated directly but requires a message to a remote processor. If an application specifies an incorrect cluster mask for a kernel object, the application either pays a performance penalty or the operation fails altogether. In either case the application is unable to corrupt the internal kernel state.

When changing the synchronization scheme (e.g., between lock-based and message-based), the kernel has to guarantee correctness during the transition period. Switching between synchronization primitives takes place in two stages. First, the new synchronization primitive is enabled while the previous remains active. Second, the previous primitive is deactivated, but only after it is structurally guaranteed that the new scheme is honored by all processors.

4.3.2 Dynamic Lock Primitive

In order to accomplish dynamic adaptability, I extend the memory-based lock primitives by an *enable state*. The enable state denotes whether or not the lock primitive is active. In disabled state, the lock operation (e.g., an atomic operation) is not executed thus incurring no (or only marginal) overhead.

One problem is that switching between enabled and disabled states has a transition period, where processors of a multiprocessor system have an inconsistent view on the enabled-state of the lock primitive. Until the transition period is completed, the kernel must preserve the *old semantic* of the synchronization primitive.

When a lock is to become disabled, the lock primitive must remain active until no processors try to acquire the lock. Similarly, when a lock is to become enabled for a critical section that was dedicated to a single processor before, no other processor can enter the critical section until the previous owner is aware of the new lock semantic.

For common kernel lock primitives it is impossible to derive whether remote processors actively try to acquire the lock. For example, spin locks are implemented via a shared memory location. A processor acquires the lock by atomically replacing the memory content with the value that denotes a locked state. If the previously stored value was locked, the processor retries the operation. It is not possible to directly derive when an update of the lock state will have propagated to all processors. A time-based scheme — disable the lock and wait x cycles — is unfeasible, because it is not possible to predict x. Even by adopting the spin loop such that it explicitly checks for a state change, it is not possible to predict memory access latencies in large multiprocessor systems. Furthermore, hardware mechanisms, such as IA-32's system management mode, eliminate any predictability.[3]

Read-copy update successfully addresses the same problem with its epoch scheme [80]. RCU passes a token between the processors; passing the token ensures that the processor executes a *safe* code path (within the kernel) that is guaranteed to hold no data references. Obviously, the RCU token thus also guarantees that a processor is not actively trying to acquire a lock. Hence, a full round-trip of the token guarantees that all outstanding lock attempts have completed and that all processors are aware of the update of the lock primitive.

RCU delays the destruction of data objects as long as references may still be held. Freed objects are explicitly tracked in a separate list; the list is processed at expiration of an RCU epoch. Such explicit tracking is unfeasible for locks due to the required memory footprint and also because locks are in most cases embedded in objects and those may get released.

Instead of externally tracking outstanding lock state updates, I integrate the state updates with the lock primitive. This scheme eliminates the need for explicit tracking and the existence problem of the object the lock is embedded in. Furthermore, only those locks that are actively used get modified. Each dynamic lock carries an RCU epoch counter that denotes when the state switch is safe. The lock code tests whether the lock is in a transition period. If the stored RCU epoch counter denotes that the epoch expired, the locking code finalizes the state switch. In addition to the *enabled* and *disabled* state, the lock has two intermediate states:

[3] System management mode is activated by a non-maskable interrupt. The processor enters a special system mode that is not under control of the operating system. The code is provided by the platform vendor and loaded at system boot time. There are no time bounds whatsoever and the platform vendor only guarantees that the system gets reactivated at some point.

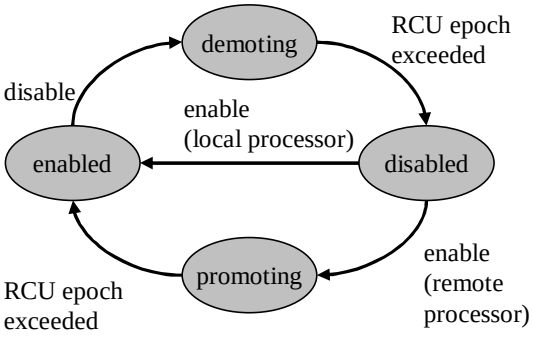


Figure 4.5: State diagram for promotion and demotion of spin locks

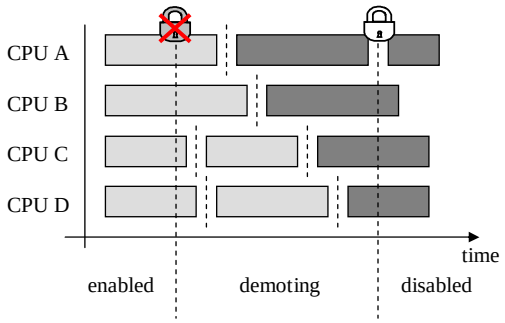


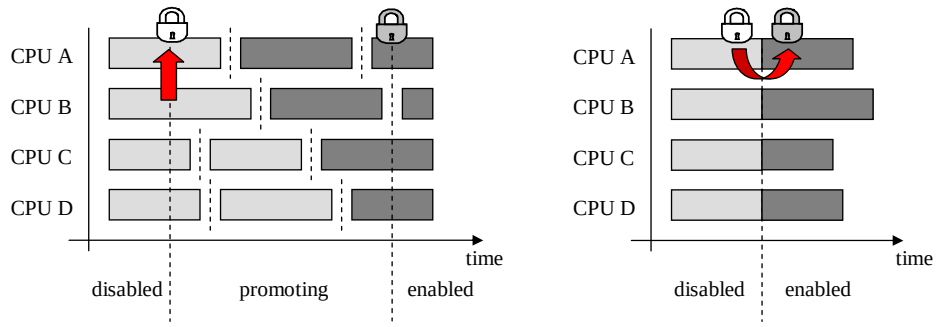
Figure 4.6: Delayed lock demotion based on RCU epoch. After disabling the lock, it has to remain active for one complete RCU epoch to ensure no more outstanding lock attempts by other processors.

promoting and *demoting* (see Figure 4.5). Only when the lock is in the disabled state, can the locking code be skipped safely.

Figure 4.6 depicts the transition from an enabled to a disabled lock state including the required transition period for the RCU epoch. When enabling a disabled lock I distinguish two cases: If the lock is disabled because the protected object is only accessible from its home processor, the algorithm must ensure that the processor is not within the critical section. If the lock is promoted on the home processor itself, it is sufficient to change the lock's state from disabled to enabled (see Figure 4.7b). However, when another processor initiates the lock promotion, the lock cannot be taken until completion of a complete RCU epoch cycle (see Figure 4.7a).

The additional functionality induces a higher overhead on the lock primitive as the lock has a higher cache footprint to encode the lock state and RCU epoch, and also a higher execution overhead for the additional checks. One can minimize the overhead by carefully locating the state and RCU epoch data and by using an efficient data encoding of the lock state.

The lock state is accessed on each lock acquisition and is thus performance



(a) Remote (b) Local

Figure 4.7: Lock promotion for a disabled lock from (a) a remote processor and (b) a local processor. The lock is restricted to CPU *A*. When CPU *B* enables the lock (a), it cannot be taken until after at least one RCU epoch in *A*. The lock promotion therefore requires two RCU epochs in *B*. When the lock is directly enabled on CPU *A* the lock can be immediately enabled because only *A* could potentially hold the lock (which it does not).

critical. The cache footprint for the state is a few bits that are mostly read and only modified when the state of the lock changes. Co-locating those bits with other accessed read-only data minimizes the cache footprint and overhead due to cache-coherency updates. Testing the state can be used to prefetch other data. The RCU epoch counter is only accessed during the state transition periods and thus in general is less performance critical.

The four states can be efficiently encoded in two bits. The encoding should be optimized for the two performance critical cases, that is when the lock is in *disabled* or *enabled* state. I specifically chose a zero value for the disabled state, since in all other states the lock operation has to be performed. On many architectures a test for zero is particularly optimized.

The following listing shows sample code for a normal spin-lock primitive. A value of zero in the lock variable (lock) denotes that the lock is free, otherwise the lock is taken. The overhead for a disabled lock using the specific encoding is a single jump instruction.

```
1 if (state != disabled) {
2   while test&set(spinlock)
3     { /* wait */ }
4   if (state != enabled && LockEpoch+1 < GlobalEpoch)
5   {
6       /* perform lock state manipulation */
7   }
8 }
9
```

```
10 /* critical section */
11
12 spinlock = 0; /* lock release */
```

The corresponding (hand-optimized) assembler spin-lock code on IA-32 is as follows. (For clarity reasons, the listing only shows the code for the uncontended case and lacks the code for the spin loop and promotion/demotion.)

```
1   mov state, %ecx
2   jecxz critical_section ;jump if ecx=0 (lock disabled)
3   xchg %ecx, spinlock
4   test  %ecx, %ecx
5   jnz spin_loop          ;lock taken -> spin
6   test $2, state
7   jnz promote_demote     ;handle RCU epoch
8 critical_section:
9   ...
10  mov $0, spinlock       ;release lock
```

The lock code has the *same register footprint* as a normal spin lock and one more data reference for the lock state. The additionally introduced code on the critical path is line 2, 6, and 7. For normal spin locks, line 1 would not need an explicit memory reference, however, the register has to be preinitialized with a non-zero value. The specific state encoding combines the state test and register initialization in a single instruction. In a disabled state the code requires one taken jump (line 2) and in an enabled (and unlocked) state three non-taken jumps. Note that the code assumes a state encoding of disabled=0 and enabled=1 (and 2, 3 for the two other states).

On architectures with a more relaxed memory consistency model (e.g., PowerPC and Itanium), the unlock operation additionally requires a memory barrier. If the overhead of a test of the lock state is lower than the cost of the memory fence, the unlock operation should be executed conditionally.

4.3.3 Lock Granularity

With a foundation of dynamic locks as the base primitive I derive a method for dynamically adjusting lock granularity. As discussed in 4.3.1, the optimal lock granularity depends on the specific workload characteristics and access pattern to an object. Fine-grained locking induces a high overhead in non-contention cases, whereas coarse-grained locking may introduce bottlenecks and limit scalability.

The decision for a coarse-grained vs. a fine-grained locking strategy depends on the specifics of the algorithm, that is *where* to lock and *what* gets protected by locks. A detailed discussion of locking strategies is beyond the scope of this work and thus I only address the general principles.

Fine-granular locking requires a divisible data structure where each part can be individually locked without compromising the overall structural integrity. The data

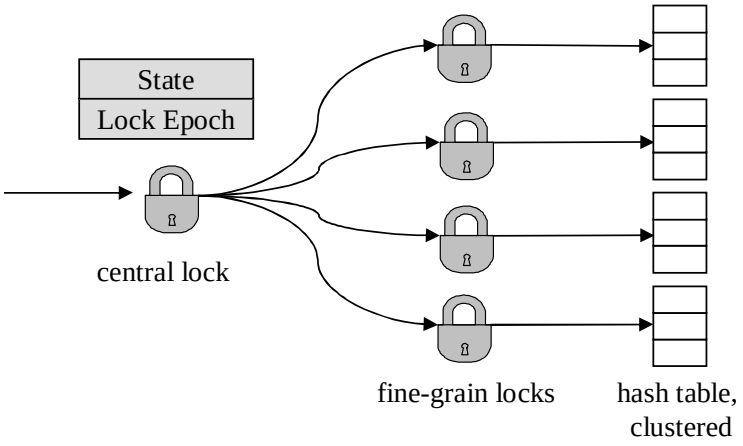


Figure 4.8: Cascaded spin locks to control locking granularity. Only one of the locks — coarse or fine — is enabled at a time. Reconfiguration at run time addresses contention vs. overhead.

structure is divided into multiple parts and each part is protected by an individual lock. The lock granularity depends on the number of independent parts of the overall data structure.

I make the lock granularity adjustable by introducing *multiple locks* to the same object — one for each locking granularity. In order to lock an object, *all* locks have to be acquired, while some locks can be disabled. Acquiring all locks ensures that processors synchronize on the same locks, whichever lock may be active at the time. Deadlocks can be avoided by simple lock ordering [106], for example, coarse-grain locks are always acquired before fine-grain locks. The lock release has to take place in reverse order of the acquisition. Figure 4.8 shows an example of multiple cascaded locks protecting a hash table and individual clusters of the table.

Switching the lock granularity for an object takes place in multiple stages. To preserve correctness, at no point in time should two processors use a different lock granularity (i.e., one uses coarse grain, the other fine grain locks). A safe transition between the lock modes therefore requires an intermediate step, where *both* locks — coarse and fine grain — are active. After that transition period, when all processors are aware of the new lock strategy, the previous lock can be disabled. The completion is tracked with the RCU epoch counter, similar to the dynamic locks.

At first, the cascade may appear overly expensive due to the additional memory footprint for the fine granular locks. However, efficient encoding and dynamic memory allocation reduces the overhead. Instead of maintaining a lock state for each individual fine-grain lock, it is sufficient to maintain the overall lock granularity for *all* locks in a single field. The memory overhead for a two-level lock cascade is therefore identical to dynamic locks: two bits for the state and the RCU

epoch counter.

Since dynamic locks only access the lock variables when the locks are enabled, the memory required for the lock variable can be dynamically allocated (i.e., when first switching to fine-grain locking). The following listing shows pseudo code for a two-level lock cascade including code that handles dynamic adjustment.

```
1  enum State = {Coarse, Fine, ToCoarse, ToFine};
2  if (State != Fine) {
3    lock (CoarseLock);
4    if (State != Coarse AND LockEpoch+1 < GlobalEpoch) {
5      if (State == ToCoarse) {
6        State = Coarse;
7        /* potentially release fine lock memory */
8      }
9      else
10       State = Fine;
11   }
12 }
13
14 for index = 0 to NumberObjects {
15   if (State != Coarse) lock (FineLock[index]);
16
17   /* critical section */
18
19   if (State != Coarse) unlock (FineLock[index]);
20 }
21
22 unlock (CoarseLock);
```

A special case of the scheme occurs when the fine-granular objects themselves are not locked but modified with an atomic operation (e.g., atomic compare-and-swap). In coarse-grain lock mode, all objects are protected by a single lock and the data members can be modified with normal (non-atomic) memory operations. In fine-granular synchronization mode, manipulations are performed with multi-processor safe atomic operations. The following pseudo code shows that scenario.

```
1  enum State = {Coarse, Fine, ToCoarse, ToFine};
2  if (State != Fine) {
3    lock (CoarseLock);
4    if (State != Coarse && LockEpoch+1 < GlobalEpoch) {
5      if (State == ToCoarse)
6        State = Coarse;
7      else
8        State = Fine;
9    }
10 }
```

```
11
12 if (State == Coarse)
13   object = newvalue;
14 else
15   CompareAndSwap(object, newvalue, oldvalue);
16
17 unlock (CoarseLock);
```

The overhead for the special handling is minor on most architectures. For example, IA32's cmpxchg instruction requires an additional instruction prefix to differentiate between the synchronized multiprocessor and the unsynchronized version of the instruction. A conditional jump over the one-byte instruction prefix, depending on the lock state, induces only marginal overhead and code complexity, while reducing the cost from 136 to 12 cycles (Pentium 4 2.2GHz). Itanium's predicates allow for a similarly efficient implementation.

4.4 TLB Coherency

In the previous section I addressed the performance aspects of coarse- vs. fine-granular synchronization in the kernel. TLB coherency updates have similar performance trade-offs that are addressed in this section.

Operating systems store memory address translations and permissions in page tables. The translations are cached in the processors' TLBs. After modifications to page tables, processors may still have old values of the modified page table entries cached in their TLB; page tables and TLBs are thus inconsistent. Consistency is recreated by invalidating the affected TLB entries. All processor architectures offer specific TLB invalidation instructions for that purpose.

When page tables are shared between multiple processors, a page table entry may be cached in TLBs of multiple processors. On most architectures, the scope of a TLB invalidation operation is limited to the executing processor. The modification of page permissions thus requires a remote TLB invalidation that is executed on the remote processor.

A shoot-down requires an IPI and interrupts the active operation on the remote processor. Remote TLB shoot-downs have a high overhead similar to message-based synchronization. The changed permissions in the page table become effective only *after* all remote processors invalidate the respective TLB entries.

In one special case — when the permissions to a page get extended — the page fault handler can avoid the necessity for the TLB shoot-down. When the page access raises a fault because of a stale TLB entry, the fault handler validates the page permissions and if correct only reloads the entry and restarts.

If completion of a system function depends on the completion of permission updates, then the remote TLB shoot-down latency directly adds to the function's cost and latency. Typical examples for such scenarios are permission updates for

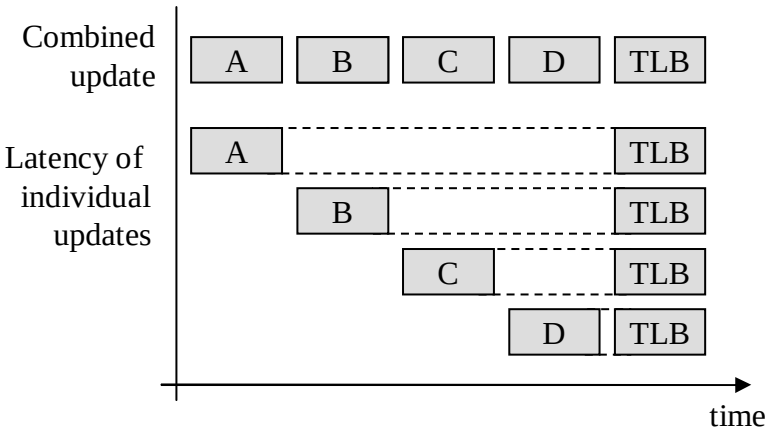


Figure 4.9: Implication of combined TLB shoot-down on latency for individual entries. The first row shows the combined operation for updating four page table entries (A–D) followed by a combined TLB shoot-down. The rows below show the latency for each individual update.

copy-on-write (as used for fork) and page swapping to disk. Both operations depend on the completion of the permission update for their semantic correctness.

I define the following two design goals for the kernel's TLB coherency algorithm:

1. The overhead for TLB shoot-downs — and thus the absolute number of shoot-downs — should be minimal. The kernel has to combine as many TLB shoot-downs as possible into a single remote invocation. Shoot-downs should be reduced to only those processors that require an update (i.e., no global TLB shoot-down).

2. The latency of independent memory updates and therefore the latency of TLB shoot-downs should be minimal.

The first design goal is achieved by combining multiple TLB coherency updates into one single shoot-down cycle after all page tables are up-to-date. However, a combined shoot-down violates the second design goal: low latency. When multiple processors concurrently manipulate memory permissions for the same memory object, the operations of those processors become inter-dependent. An update of page permissions is completed after the completion of the TLB coherency cycle (i.e., when all remote TLBs are valid). By combining multiple TLB shoot-downs in one, the duration of the permission update is the length of the *overall operation* for all updates (see Figure 4.9).

In order to ensure correctness, permission updates have to be synchronized between all processors. When one processor changes the memory permissions

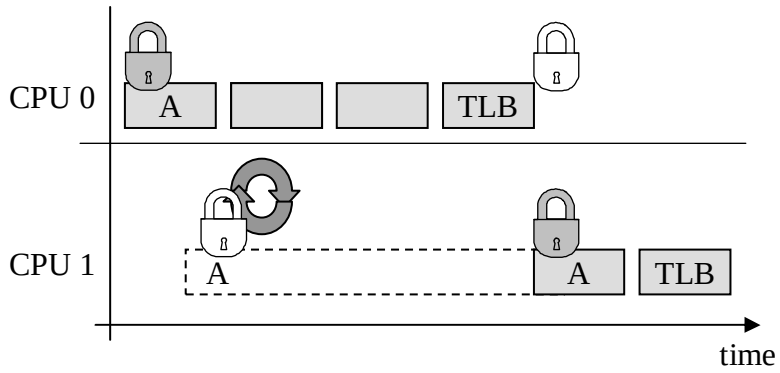


Figure 4.10: Concurrent permission update on two processors. CPU 0 updates the permissions for multiple pages including *A*. In order to guarantee consistency it locks *A* and releases the lock after the TLBs are coherent. When CPU 1 tries to manipulate *A* shortly after the lock acquisition by CPU 0, it has to wait until completion of the operation on CPU 0.

but postpones the shoot-down, a second permission update by another processor must not complete before the first shoot-down is completed. Otherwise, the second processor incorrectly assumes active page permissions (e.g., as required for copy-on-write) that are not yet enforced. The inter-dependency is a performance problem because inexpensive permission updates that only require a simple memory access suddenly become long-running operations that also include the latency for remote TLB shoot-downs. Figure 4.10 depicts the delay for a concurrent operation that depends on the completion of another processor's update. The inter-dependency is also a scalability problem, because operations that are independent and may be processed in parallel are now serialized.

4.4.1 Independence of Coherency Updates

The delayed TLB shoot-down dynamically creates coarse-grain memory objects, although individual memory objects may have fine granular locks. The problem is the tight coupling of modification of memory permissions with the TLB shoot-down. The necessary TLB shoot-downs are derived and accumulated when updating the page table. That information is only available to the processor that modifies the page permissions creating the aforementioned inter-dependency. Remote processors become dependent on the completion of the operation, because they are *not aware* what TLB shoot-downs are still outstanding.

To counteract the scalability limitations induced by the combined TLB updates, I propose an algorithm that eliminates inter-dependencies between page permission updates and TLB shoot-downs. The previously described TLB consistency scheme is based on a strict causal order; operations are globally ordered and thus safe. I

relax the ordering model by separating page table updates from the TLB coherency updates. The algorithm works as follows:

The modification of memory permissions requires a corresponding TLB update. Only after the update completes are the permissions active and globally enforced. When all required TLB shoot-downs have completed, an operation that relies on the page permissions stored in the page table is safe. However, that requirement is *independent of a specific processor*; the TLB shoot-down operation can be initiated by any processor in the system.

In order to detect still outstanding TLB updates, I introduce a per-processor TLB update epoch. When a processor updates memory permissions to a memory object, it tags the object with the processor ID and the processor's current TLB epoch. Furthermore, the processor records the necessary TLB updates — thus includes all remote processors and TLB entries that need to be invalidated (details follow in 4.4.2). A new TLB epoch starts after all shoot-downs are completed.

When a second processor updates the memory permissions to the same object, it compares the stored epoch counter with the corresponding processor's current TLB epoch. When they are different, the TLB shoot-down cycle is complete and the page table permissions are authoritative. However, when the stored epoch is identical to the stored processor's current epoch, the TLB update is still outstanding and the TLB shoot-down is initiated by the second processor.

To preserve the ordering of operations, the second processor updates the processor ID and epoch counter stored with the memory object. Furthermore, it includes the recorded TLB entries into its own shoot-down list and updates the list according to the performed updates. Thus, if a third processor updates the permissions of the memory object, it now becomes dependent on the second processor's TLB epoch. The formal presentation of the algorithm is as follows:

Let D be the set of TLB entries t_i that need to be invalidated, with $t = (v, p, a)$. v denotes the virtual address of the entry that is invalidated, p the processor that may cache the entry, and a the address space identifier for the particular processor. Let o be the memory object for which the permissions are changed, with $o = (D, e, p)$. $o.e$ denotes the TLB epoch $p.e$ of processor p stored in o, and $o.D$ denotes the set of dirty TLB entries, with $o.D = \{t\}$. When changing the memory permissions for object o that requires to invalidate D TLB entries, then $o.D' = D$ if $o.e \neq p.e$ or $o.D' = D \cup o.D$ otherwise.

4.4.2 Version Vector

The described algorithm stores the TLB entries that need to be invalidated with the object. The required storage memory depends on the number of affected TLB entries, the number of processors that concurrently manipulate the object, and the frequency of updates (i.e., whether updates complete or many TLB shoot-downs are outstanding).

A naïve tracking algorithm that only records the dirty TLB entries, has an unbounded memory consumption and is therefore unfeasible. I reduce the memory

requirements by incorporating architectural knowledge about the MMU. While the TLB epoch tracking is a general mechanism and applicable to all hardware architectures, the tracking of outstanding TLB shoot-downs depends on very specific details of the MMU, cache, and TLB. The influential properties include whether or not the TLB is tagged, the TLB shoot-down primitives and their cost, the overhead for TLB refills and also the cache line transfer costs and the latency of an IPI. A detailed analysis for a wide variety of MMUs is beyond the scope of this work, and thus I only provide an exemplary validation and discussion for my reference architecture: Intel's IA-32.

IA-32's MMU has dedicated data and instruction TLBs with a hardware-walked two-level page table.[4] An address-space switch is realized by loading a different page-table base pointer into the processor's CR3 control register. The architecture does not define address-space identifiers and all (non-global) TLB entries are automatically flushed on every CR3 reload. An update of CR3 therefore serves two purposes: (*i*) to switch between virtual address spaces and (*ii*) to invalidate the TLB. The architecture additionally provides the invlpg instruction for selective invalidation of individual TLB entries.

A TLB shoot-down can be either performed by multiple invocations of invlpg or via a complete invalidation. A complete invalidation has the disadvantage that it also invalidates entries that are still in use and thus need to be reloaded. On the other hand, individual invalidations have a significant overhead (apparently, invlpg is not optimized). In Figure 4.11 I compare the overhead for individual shoot-downs against a complete invalidation plus the cost for the re-population of the TLB. The break-even point for the used test environment (a 2.2 GHz Pentium 4 Xeon) is at eight TLB entries. Starting with nine entries it is more efficient to invalidate the complete TLB rather than invalidating individual entries. Obviously, the specific break-even point depends on the architecture's TLB size and overhead for shoot-downs.

The lack of address-space identifiers reduces the amount of data that needs to be tracked. The dirty TLB entries are represented by $t = (v, p)$. On modification of permissions of a page v in a page table that is shared across a set of processors $p_0, ..., p_{j-1}$, the set of dirty TLB entries is $D = \{(v, p_0), (v, p_1), \ldots, (v, p_{j-1})\}$. The memory footprint of a simple encoding of the set is $|D| \cdot \text{sizeof}(t)$ which does not scale for many processors. Thus, the algorithm requires a more compact representation.

The TLB shoot-downs ensure that no stale entries are cached in remote TLBs. Most TLB replacement algorithms are indeterministic and a TLB shoot-down may target a TLB entry that is not in the TLB anymore or even never was. A shoot-down strategy still preserves functional correctness if it targets *more* processors. All stale TLB entries will definitely be evicted, however, the additional (and unnecessary)

[4] At the time of writing, IA-32 defines four different page table formats. For this work I only consider the standard 32-bit virtual and physical addressing mode with support for 4MB super pages and the global tag [29].

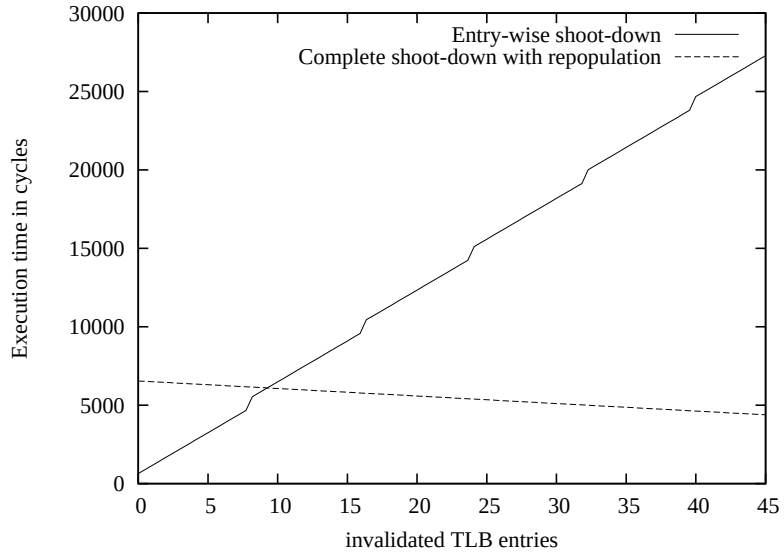


Figure 4.11: Comparison of TLB shoot-down costs with complete re-population against an entry-by-entry shoot-down for a Xeon Pentium 4 2.2 GHz with 128 TLB entries. The costs include the overhead for the invalidation instruction (521 cycles) and required cache line transfers (about 80 cycles). The break-even point is at eight entries. The negative slope of the repopulation cost curve reflects the decreasing number of repopulated entries for an increasing number of shoot-down values. Note that the overhead for the required IPI is not shown.

shot-downs may incur a higher cost.

Based on that observation, the set $D = \{(v, p_0), (v, p_1), \ldots, (v, p_{j-1})\}$ can also be encoded as the tuple (v, m), whereby m denotes the processor cluster mask (as defined in Section 4.2.2). The shoot-down is correct if the following condition holds: $\forall i : (\exists t \in D : t = (v, p_i)) \Rightarrow f(m, i) > 0$. The cluster mask's fixed size reduces the memory footprint for tracking of an updated page table entry from $|D| \cdot \text{sizeof}(t)$ to $\text{sizeof}((m, v))$ where $|D| < n$. Note that the limitations of the memory bus (such as NUMA) make highly shared memory resources unlikely, and therefore the encoding via the cluster mask is accurate for the vast majority of cases (see Section 4.2).[5]

The manipulation of memory permissions of an object may require a TLB shoot-down of n entries $D = \{(v_0, m_0), (v_1, m_1), \ldots, (v_{n-1}, m_{n-1})\}$. The memory footprint depends on the maximum size of the object (and thus the maximum number of potential dirty TLB entries). The overhead of the by-entry shoot-down operation of the processor architecture limits the number of entries that need to be recorded. As shown in Figure 4.11, the cost for an entry-by-entry TLB invalidation exceeds the cost for a complete invalidation for more than eight entries. Hence, the

[5] Note that accuracy addresses the encoding of processors, not the accuracy of the algorithm itself.

hardware limits the memory footprint to $max(|D|) = 8$ entries. Afterward, a complete TLB invalidation becomes more efficient.

The number of remote shoot-downs can be further reduced by testing whether or not the currently active remote page table (i.e., current CR3 value) corresponds to the address space of the tracked TLB entry. If the processor loads a new value into CR3 it implicitly invalidates all TLB entries and the explicit shoot-down can be omitted. However, the test misses the scenario where CR3 is reloaded multiple times and matches the recorded value. The simple equality test delivers a false positive and indicates a required TLB shoot-down although the TLB is already up-to-date.

Instead, I introduce a *TLB version* that increases on every CR3 reload (including context switches and TLB invalidations). The version significantly decreases the probability for a false positive. After modification of memory permissions in page tables, the algorithm stores the TLB versions for all affected processors in a vector $w = (w_0, w_1, \ldots, w_{n-1})$. If the remote TLB version already increased at shoot-down time, the shoot-down is superfluous and can be omitted. The maximum size of the vector depends on the number of processors and requires one vector per memory object.

The last optimization is to reduce the number of vectors to one per processor at the cost of some additional TLB shoot-downs. At system startup time the kernel preallocates a version vector per processor. The version vector also contains the storage space for the individual entry-by-entry shoot-downs. Memory objects contain (1) a reference to the version vector of processor p, (2) a processor cluster mask $o.m$, and (3) the TLB epoch $o.e$ — an overall memory footprint of three processor words per memory object.

After modification of page tables, the version vector needs to reflect the required TLB shoot-downs (including those still outstanding from other processors). If the TLB epoch that is stored with the object is still active ($o.e = p.e$), the shoot-downs of the previous processor did not yet complete. In that case the algorithm updates the current processor's version vector by incorporating the still outstanding shoot-downs: $w' = (w'_0, w'_1, \ldots, w'_{n-1})$ with $w'_i = max(w^1_i, w^2_i) \forall i \in (0, 1, \ldots, n-1) \mid f(o.m, i) > 0$. Additionally, the algorithm updates $o.e$ with the current processor's TLB epoch $p.e$, the reference to the processor $o.p = p$, and the cluster mask $o.m$.

The described optimizations drastically reduce the memory footprint making TLB tracking feasible. The reduced memory footprint is at the cost of accuracy and potentially results in a higher overhead compared to the optimal case. However, the TLB version vector can eliminate many unnecessary remote notifications via IPIs, if the remote processor already performed a CR3 reload — either due to a concurrent TLB shoot-down from another processor or by normal context switch activities. As initially emphasized, the solution is specifically optimized for IA-32's MMU and because of this other architectures may have different cost-performance trade-offs.

4.5 Event Logging for Application Policies

In uniprocessor systems, scheduling is the activity of deciding which thread of control gets to run on the CPU. In multiprocessor systems, scheduling has another dimension, not only deciding when a thread will run but also *where* it will run, (i.e. on which processor). In order to make informed resource scheduling decisions (not limited to thread allocation), a *dynamic scheduler* needs to combine information from a variety of sources. The vast majority of scheduling algorithms (naturally) assume data availability of a monolithic system structure; a single entity maintains all system states and provides full accessibility and visibility.

The common definition of scheduling includes two dimensions: time and locality. Although time and locality are inter-dependent they can be handled autonomously. In the following, I will primarily focus on the aspect that is specific to multiprocessor system: the allocation of threads to processors.

Fundamental to microkernel-based systems is a divided system structure with highly distributed information that is required for resource scheduling. A scheduler needs to evaluate runtime information and therefore needs access to runtime data in the first place. Strict isolation and independence creates a boundary that obstructs the free information flow available in monolithic systems.

The major challenge is that resource scheduling takes place sporadically and infrequently, however, the scheduling decision requires up-to-date information. Efficient scheduling requires sufficiently accurate data that — in most cases — is never evaluated by the scheduler. Direct access to system-internal data structures minimizes the overhead in monolithic systems, whereas a microkernel system requires additional action to extract, accumulate, and provide that data. Hence, isolation increases constant runtime overhead for system components as well as the kernel and therefore decreases the overall system performance.

To address this problem, I developed an efficient mechanism for runtime data aggregation with a high-bandwidth interface to a resource scheduler: shared memory. A primary design goal is to have a single mechanism that is applicable to the microkernel as well as to system components.

The section is structured as follows: First, I discuss the properties of data that are the basis for scheduling decisions. Afterward, I describe mechanisms to collect and deliver scheduling-relevant data to a scheduling component. While system-level components can be freely adapted (and potentially replaced at runtime), the microkernel is static. The wide variety of existing resource scheduling algorithms requires a flexible mechanism that covers as many usage scenarios as possible. While very detailed event logging can most likely fulfill that requirement, it incurs a prohibitively high runtime overhead on performance-critical code paths. I therefore develop an adaptable event logging scheme that can be dynamically enabled and adjusted minimizing the overall performance impact on the system.

4.5.1 Overview

Resource scheduling policies base their decision on system runtime information, such as the current load situation [37] derived from the run-queue length [14, 96], and communication pattern between tasks [36, 88, 99]. An important performance aspect for scheduling is the resource affinity of processes, such as active cache working sets or NUMA memory [16, 17, 107].

Efficient resource allocation and scheduling in multiprocessor systems is a widely researched area. A detailed discussion of individual resource scheduling policies or comparison is beyond the scope of this work. For a detailed discussion I refer to an excellent survey of parallel job scheduling strategies for multiprocessor systems by Feitelson [35]. In this work, I solely focus on the *required mechanisms* for information gathering in order to realize scheduling algorithms.

Dynamic schedulers try to extrapolate future system behavior from past behavior. Hence, they require statistical data on system events in relation to the schedulable principals and system resources. A naïve call-back on event occurrence can provide very detailed and timely information to the scheduler. In a monolithic system, a callback is a simple function call, whereas in a microkernel-based system a callback may require multiple privilege level changes and address space switches. Thus, callbacks induce an unacceptable runtime overhead for frequent events.

The *acceptable overhead* for runtime data collection has to be considered for two scenarios. First, the overhead of a scheduling algorithm is obviously bounded by the maximum achievable benefit. If the potential performance benefit is less than the overall cost for data aggregation for deriving a scheduling decision, a random allocation policy is preferable. The second relevant performance metric is the cost for steady state after an optimal resource allocation is reached. Here, the algorithm should only incur a minor runtime overhead.

The upper bound of the frequency for scheduling decisions is given by the overhead induced by the resource reallocation. The costs fall into two categories: the cost of the operation and the follow-on costs. Applications benefit from the migration only after the overhead for both categories are amortized. In today's multiprocessor architectures, the follow-on costs induced by cache working set migrations are the dominating performance factor. The time spans between re-balancing and migration decisions are therefore often in the order of multiple milliseconds.[6]

4.5.2 Data Accumulation

Event recording is an effective method for offline or postponed system analysis. On event occurrence, the system writes a log entry that sufficiently identifies the event source, event reason, relevant system state, and point in time. Later, the system behavior and causal dependencies can be reconstructed from the logged in-

[6] The load balancer of Linux 2.6.10, for example, makes re-balancing dependent on the cache migration overhead. The minimal re-balancing period for SMP processors is 2.5ms whereas it is 10ms for NUMA nodes.

formation. Event logging is primarily used for debugging purposes and bottleneck elimination [121].

I apply the fundamental idea of event recording, but as a method for *online data collection* for user-level schedulers. The microkernel and system components record system events as log entries into a memory-backed log space. Shared memory provides a low overhead access method to system events for the scheduler while preserving strict separation between the individual components (i.e., the producer and consumer) [42].

The sheer amount of events and event sources renders a simplistic per-event logging scheme unfeasible. However, with increasing frequency of event occurrence, the carried information per event decreases. Scheduling decisions are in many cases based on a quantitative analysis (such as number of events per time interval [35]). In those cases the cost can be reduced via data aggregation, for example only counting the number of events instead of logging individual event instances.

The overhead for logging falls into three primary categories: (*i*) overhead for the event recording code, (*ii*) overhead for data analysis by the scheduler, and (*iii*) overhead induced by higher cache footprint for the logged information. The overhead of event recording can be reduced by the following three aggregation methods:

Less frequent logging of the same event. Combining multiple event occurences in a single log entry reduces the frequency of logging and thus the runtime overhead. The minimal execution overhead induced by event logging is an arithmetic add (for event counting) followed by a conditional branch into the event logging code. With more events combined in a single log entry, the lower becomes the overhead for logging. The overhead for event counting itself is minimal (the cache footprint of the counter and two instructions).

Logging of less event sources. The second method is to reduce the event sources, which can be achieved by conditional logging and filtering. Depending on the scheduling policy, not all system events are relevant and get evaluated. Conditionally logging of only the relevant events reduces the static runtime overhead. Filtering allows more flexibility for event selection than a simple on/off scheme, however, a complex filtering scheme may induce a higher runtime cost.

Logging of less information per event. Finally, the overhead can be reduced by limiting the amount of log data and thus the overhead to generate the log entry itself.

Event aggregation automatically reduces the overhead for data analysis in the scheduler. The scheduler handles already compacted and preprocessed log entries. Similarly, filtering eliminates irrelevant data from the log file and thus reduces the runtime overhead for data analysis. Using alternative log buffers per event source

reduces the event parsing overhead and also the memory and cache footprint. Here, the scheduler can implicitly derive data from knowledge about the log buffer.

The memory and cache footprint for event logging depends on the frequency of events, frequency of log entries, and the log entry size. In many cases, statistical analysis considers only a single or very few entries that happened last. A circular log buffer, which is exactly sized for its required back log, can minimize the overall cache footprint.

4.5.3 Event Sources and Groups

Event sources differ in their expressiveness and potentially require additional context information to become useful for a scheduler. I identified the following three primary event types:

Access, modification, or occurrence. This event type denotes the occurrence of one specific event, such as the access to a resource or the execution of a specific function. It is unrelated to other events in the system. Event logging may either log the particular occurrence or simply increase an event counter.

The event has four parameters: (*i*) a resource identifier, (*ii*) an identifier of the resource principal, (*iii*) the number of accesses, and (*iv*) a time stamp. The resource identifier uniquely identifies the accessed resource and the resource principal identifies the subject that performed the access. Based on the logged resource principal, a scheduler can account resource usage to a system's accounted subjects, such as threads or tasks.

Entry and exit. In time sharing systems, resource utilization is accounted to resource principals during the time of activity. The relevant information for counted events, such as resource usage, time, and hardware performance counters, is the number of events from activation until preemption. It can be derived by recording the absolute value of an event counter at the point of activation and at the point of preemption. The difference of both values provides the number of events during the time of activity.

Compared to on-access logging, entry–exit logs have a significantly lower log footprint and additionally cover asynchronous event counters, such as hardware-provided performance counters. Entry–exit logging combines the preemption event of a principal with the system information (i.e., counters and time).

An entry–exit event has four parameters: (*i*) a resource principal, (*ii*) an event counter, (*iii*) the new value, and (*iv*) a time stamp.

Asynchronous events. Asynchronous events are independent from a resource principal and execution. On event occurrence, the system logs the system state and point in time. A typical example for this event class is a periodic

interval timer or a hardware event counter. Asynchronous events have the same parameters as on-access events.

Events can be grouped together if at least one event parameter is identical. In that case the specific parameter can be omitted in the log. Specific event grouping depends on the event type and what values are useful for a particular scheduling policy. For example, time stamps may not be required for a scheduler that performs periodic sampling.

Entry–exit events are bound to resource principals. The primary resource principals in systems (e.g., threads or tasks) may not sufficiently represent inter-dependencies, such as client-server relations or act-on-behalf. While fine-granular resource principals allow for a detailed analysis, they also increase the number of entry–exit events and thus log entries.

Banga et al. [13] propose resource containers for separating traditional first-class resource principals from resource accounting. Instead, resource usage is accounted to a shared container with multiple associated threads or tasks. The concept of grouping resource principals in orthogonal resource accounting domains is an efficient method for event filtering as well.

4.5.4 Log Entries and Log Configuration

Event logging is on the critical code path and low overhead is therefore of paramount importance. While at application level the application designer has the freedom to tune the log structure and log values toward one or a few specific scheduling policies, the microkernel lacks such freedom. The kernel's static code base requires runtime configuration of the log facility that is flexible but still incurs a marginal overhead.

Runtime configuration for performance and event tracing is well-known for microprocessors. Most modern microprocessors support hardware performance monitors (HPM) that provide statistical information on hardware events [29, 30]. HPMs are used primarily for performance and bottleneck analysis [22, 103], but have also been proposed as a data source for scheduling decisions [120]. HPMs are configured via control registers and count hardware event occurrences. More sophisticated performance monitoring facilities, such as Intel's Pentium 4, even provide event logs.

Similar to HPMs, I introduce kernel *event-log control fields*. Kernel events are associated to control field that allow for precise control of the generated event log entries including deactivation of the log facility altogether. The control field is constructed for maximum flexibility of statistical analysis with low overhead. Each control field consists of the following set of parameters:

Log pointer and log size. The log is a circular ring buffer denoted by a log pointer and a fixed size. The size is encoded as a bit mask that is logically AND-ed to the log pointer. By using a mask it is possible to arithmetically derive the

start and the size of the log buffer. This way buffer start, buffer size, and the current log index is encoded in only two fields.

The encoding restricts the possible log buffer alignment but simplifies overflow checks. Each entry has a fixed size that depends on the logged values per event. A simple offset calculation delivers the next entry. A log buffer with a *single entry* serves as an event counter.

Overwrite or count. The scheduler can specify if event occurrence should overwrite the log event counter or add the counter to the previous value in the log. This selector enables event tracing (like the time of the last event) or event counting.

Time stamp and time source. A system may provide multiple time sources and the cost for recording a time stamp may differ. While the processor cycle counter provides a highly accurate time source, the overhead is significant on some architectures.[7] Alternatively, the system may use a less accurate interval timer, or a simple counter if the time stamp is solely required to derive causal dependencies. For many events the time of the event is irrelevant (or known) and logging the time stamp can be avoided altogether.

Threshold. For very frequent events a per-event log is too expensive. Therefore, each control register additionally contains an event counter that is incremented per event. A counter overflow leads to recording of the event and a reload of the counter with the programmed threshold value.

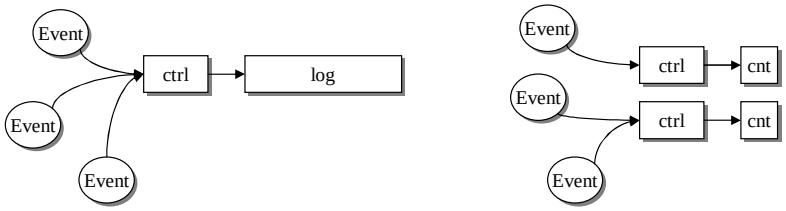
For higher flexibility, event sources and thresholds are decoupled from the log buffer. Multiple event sources can be freely associated to either one event-log control registers or alternatively to individual logs. Per-processor control registers preserve independence between processors. In order to scale, there is no synchronization between processors and the log buffers are located in processor-local memory.

Figure 4.12 depicts two alternative log configurations, one combined log and a second configuration using the event log as a counter.

4.5.5 Log Analysis

Having a detailed log of resource information, schedulers further need to evaluate the logged data. The structure with a shared memory log makes the logging facility a classical producer–consumer problem, with the logger being the producer and the scheduler being the consumer (as shown in Figure 4.13). When a scheduler processes a log file, new entries may be written concurrently, potentially leading to a scheduler evaluating inaccurate data. I discuss possible counter-measures against evaluation of inaccurate data, with a focus on minimal runtime overhead for both

[7] The Intel Pentium 4 architecture has an overhead of 125 cycles for a time stamp counter read.



(a) Combined event logging (b) Event counting

Figure 4.12: Event log control. Events refer to a log control structure that refers to a log buffer, log entry count, and the logged values. Multiple events can be logged into the same log buffer (a). A single-entry log provides a simple counter or most recent event value (b).

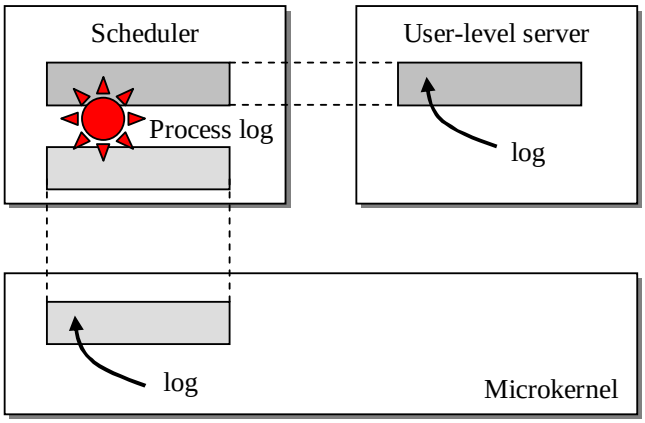


Figure 4.13: Log analysis via shared memory. The microkernel and a user-level server log events into memory that is shared with the scheduler. The scheduler processes the logged entries and derives its scheduling logic.

producer and consumer. The runtime overhead of atomic operations renders explicit synchronization via locks unfeasible.

One can differentiate between two event types: (*i*) infrequently occurring events and (*ii*) frequently occurring events. For infrequent events, the probability of concurrent logging and analysis is low. Either a repeated analysis run or a copy-out strategy into a separate buffer eliminate the race condition.

Frequently occurring events are the basis for statistical analysis and individual event occurrence is of less importance. Missing one of the frequent events therefore will have insignificant implications on the overall scheduling policy. In order to avoid explicit locking, the analysis algorithm repeats the log analysis run in case the log file changed during the run. However, repetitive analysis is an unbound operation and thus may lead to a lifelock where the consumer constantly updates the log and the consumer never completes its operation. The lifelock can be avoided

by careful system construction.

I derived the following lock analysis strategy for frequent events. The logger first writes the log values and afterward updates the index pointer into the log file. The analysis code ignores the current entry field because updates may still be in-flight and the data may be invalid. At the end of the analysis the code validates whether the index into the log file remained constant and if a wrap-around took place. The following listing shows sample code for the analysis algorithm.

```
1  int OldIndex, OldTimeStamp;
2  do {
3    OldIndex = LogCtrl.Index;
4    OldTimeStamp = Log[OldIndex].TimeStamp;
5
6    for (int Index=1; Index<LogCtrl.Entries; Index++)
7      Analyze(Log[(Index + OldIndex) % LogCtrl.Entries]);
8
9  } while(OldIndex != LogCtrl.Index ||
10         Log[OldIndex].TimeStamp != OldTimeStamp);
11 Reschedule();
```

The lifelock occurs in case the runtime of the analysis is longer than the time difference between two log entries. This is critical when events occur *and* get logged at a high rate. Since event logging is triggered by code execution, it either requires *preemption of the analysis code* on the same processor or *parallel execution of analysis and log code* on different processors. In order to avoid cache migration overhead, processor-local log analysis is preferable over remote analysis.

Due to the inherent performance degradation of long-running analyses (as argued in Section 4.5.1), log analysis that requires a full time slice is impractical. In the unlikely case of a preemption during the analysis run which additionally creates new log entries, a simple retry is sufficient and the next run will most likely complete. Additionally, preemption precautions that inform the kernel of a critical section (e.g., provided by L4, K42, and Raven) can further reduce the likelihood of an undesired preemption. This scheme is sufficient, as long as events are analyzed on the same processor on which the events occur and are logged.

When using a scheduler that accesses logs of a remote processor, the likelihood of a analysis-log conflict increases. For logs with only very few entries or when the event frequency is sufficiently low, a simple retry method is sufficient. However, when the time interval between two events on the remote processor is shorter than the runtime of the log analysis, the algorithm needs to take further precautions. One possibility is to perform a partial analysis and to combine the different parts at the end. Alternatively, event logging can be explicitly deactivated until the analysis completes. This approach avoids corruption of log data at the cost of accuracy. A more detailed discussion on the design alternatives and trade-offs for efficient log analysis is given in [100].

4.6 Summary

In this Chapter I described four mechanisms for adjusting kernel synchronization primitives in order to reflect the degree of parallelism of resource sharing. It is founded on an efficient tracking scheme for parallelism (Section 4.2) — the *processor cluster mask* — which allows every object to be tagged with a space-efficient processor bitmap. The cluster mask uses an encoding scheme that trades accuracy for space with increasing number of encoded processors, similar to floating point encodings.

I presented dynamic locks, which can be enabled and disabled at runtime in a safe manner (Section 4.3). Dynamic locks overcome the overhead of a single synchronization strategy (lock vs. message based) in the kernel. Using the information on parallelism encoded in the cluster mask, the lock primitives can be dynamically adjusted. Dynamic locks further allow for safe and dynamic adjustment of the *locking granularity* (Section 4.3.3). In case of no concurrency, coarse-grain locking yields better performance and reduces the cache footprint for the lock variable. In case of high concurrency, fine-granular locking reduces lock contention, increases parallelism, and thus yields better performance. The best strategy depends on the specific workload, and a static common-case selection can result in suboptimal performance. By cascading multiple dynamic locks, the synchronization strategy — coarse-grain or fine-grain locking — can be dynamically chosen and adopted at runtime.

In Section 4.4 I presented a TLB coherency scheme that enables a kernel to batch expensive TLB shoot-downs for multiple memory permission updates. Instead of updating TLBs on a per-entry basis, multiple updates are combined. While this scheme reduces the remote-processor signaling overhead, it also introduces inter-dependencies of independent and parallel operations in the memory subsystem. I developed a TLB epoch mechanism that allows processors to safely bypass outstanding TLB updates by *detecting* non-completed shoot-downs and initiating the shoot-down themselves.

Finally, in Section 4.5 I presented an event-logging mechanism for system components including the microkernel itself. Different to monolithic systems, where resource usage information is directly available to the OS, are scheduling-relevant information distributed and isolated between the system components. Event-logging with shared memory provides a high-bandwidth and low overhead channel between resource managers and the scheduler, thereby efficiently overcoming the additional isolation restrictions.

In the next chapter I describe the application of those mechanisms to a specific microkernel.

Chapter 5

Application to the L4 Microkernel

In this chapter I describe the application of multiprocessor performance adaptation to L4Ka::Pistachio, an L4 microkernel. L4Ka::Pistachio is the basis for a variety of research projects, including the development of multi-server systems [42], and para-virtualization efforts [69]. I and colleagues [111] have shown that L4Ka::Pistachio is able to efficiently run multiple instances of a multiprocessor Linux kernel on a single multiprocessor system.

L4 is an ongoing research effort and therefore a moving target. In this thesis, I refer to the general concepts of the latest L4 specification, Version X.2 Rev. 6 [49], which I co-authored. Furthermore, I extended the specification to address scalability aspects. L4Ka::Pistachio is a group effort by a number of people, however, I implemented a significant part of the main kernel and maintain the IA-32 specific part of the kernel.

In this chapter, I concentrate on aspects of L4Ka::Pistachio that relate to the main subject of this thesis: scalability of microkernel-based systems. I have described other aspects — including reasoning for individual design decisions — in a technical report that accompanies this thesis [109].

The chapter is organized as follows. In Section 5.1, I give a general overview of the core L4 concepts that are required in later sections. In Section 5.2, I define my requirements and general goals for a scalable kernel. This is followed by three sections that describe the specific application of the ideas of performance adaptation to L4Ka::Pistachio. In Section 5.3, I detail the construction of the inter-process communication primitive. Section 5.4 describes the application of dynamic lock granularity (developed in Section 4.3.3) and the TLB coherency via tracking (Section 4.4) to L4's memory management mechanisms. In Section 5.5, I describe the mechanisms for efficient resource tracking that provide the basis for user-level resource managers.

5.1 Overview of L4

L4 provides two abstractions in addition to bare hardware: *threads* and *address spaces*.

Threads are the abstraction for an activity; processor time is multiplexed between threads. A thread is represented by processor state (register context), a unique global identifier, and an association to an address space. Each thread belongs to exactly one address space at any time. In L4, threads are also the fundamental abstraction for controlled transfer of execution between different protection domains (i.e., address spaces).

Address spaces provide the abstraction for protection and isolation; resource permissions are bound to address spaces. Address spaces are passive objects that are manipulated by threads. Address spaces have no explicit names, but are named via a thread associated to the particular space.

Permissions in L4 are bound to address spaces; all threads within an address space have the same rights and can freely manipulate each other. This model has implications and significant limitations on multiprocessors discussed later in more detail.

In L4, the address-space concept is used for different resources, including memory and I/O ports.

Furthermore, L4 features two mechanisms for permission control: *IPC* and *resource mapping*.

IPC is the mechanism for controlled transfer of data, resource permissions, and control of execution between exactly two threads. If the threads reside in different address spaces, IPC implicitly crosses protection domain boundaries and transfers data between them.

IPC is a synchronous message transfer and requires mutual agreement between both communication parties in the form of a rendezvous. During the rendezvous, the message content is transfered from the sender to the receiver.

Resource mapping is the mechanism for resource durable permission delegation. Resource permissions are granted via the *map* operation, that is part of IPC and thus also requires mutual agreement. Map transfers the resource permissions from the sender's address space to the receiver's address space. The permissions to the resource are either identical to the sender's permission or more restrictive.

The map operation can be applied recursively. Permission revocation is performed via the *unmap* primitive and is involuntary for the receiver of a memory mapping. Unmap is a recursive operation which revokes all dependent mappings as well. A more detailed description follows in Section 5.4.

Permissions are associated with an address space; any thread of an address space can manipulate (and potentially revoke) resource permissions.

L4 features an in-kernel priority-based round-robin scheduler that allocates time to individual threads. When the time slice of a thread expires, the kernel preempts the thread and chooses the next runnable thread within the same priority level. When a priority level becomes empty, the kernel considers the next higher level. If no more runnable threads remain in the system, the kernel switches the processor into a low-power state.

5.2 Requirements

In this section I describe the design goals (and non-goals) for a multiprocessor version of the L4 microkernel. The description has a specific focus on the multiprocessor aspects. Afterward, I describe the extensions and modifications I make to the original uniprocessor kernel design.

The intricate interdependencies between nearly all kernel subsystems requires a good understanding and detailed discussion of many individual aspects of kernel components. I discuss them in more detail in a separate report [109].

5.2.1 Design Goals

I define four major design goals for L4Ka::Pistachio: scalability, performance, policy freedom of new mechanisms, and compatibility and transparency. Additionally, I define the following set of non-goals (i.e., aspects that were of no interest even though addressed by important related L4 work): real time [58], fine-granular user-level memory management [52, 74], and requirements of highly secure systems [60]. For all design decisions I clearly favored goals over non-goals.

Scalability

Scalability is the primary goal of this work. Following Unrau's design principles [113] for scalable operating systems, the microkernel has to fulfill the three construction principles:

1. *Preserve parallelism.* The parallelism of applications must be preserved throughout the microkernel; requests from independent applications have to be served independently and in parallel.

2. *Bounded overhead.* The overhead of operations must be bounded and independent of the number of processors.

3. *Preserve locality.* The kernel has to preserve the locality of the applications. From this requirement I derive that (*i*) the kernel code and data need to be local to the processors, (*ii*) upon application migration the kernel meta data

has to be relocated, (*iii*) applications and OS services that manage memory resources need to be aware of processor locality, and (*iv*) require mechanisms to explicitly manage it.

Performance

The performance of microkernel primitives is paramount, with a particular focus on the IPC primitive. I closely follow the design principle for microkernels as postulated by Liedtke: *Ipc performance is the Master. Anything which may lead to higher ipc performance has to be discussed. [...]* [70].

I evaluate the achievable performance based on two factors. First, the uniprocessor version of the kernel gives the upper performance bound for operations that are local to one processor. For all critical operations I choose a structure identical (or almost identical) to the single processor kernel. Second, for operations that are either cross-processor or require concurrency precautions (i.e., locking) I target for the achievable minimal overhead of the *macro operations*. A macro operation thereby combines multiple smaller and potentially independent operations into one larger operation.

The optimization at the macro-operation level allows the kernel to reduce costs by combining the startup overhead of multiple micro-operations, for example via critical-section fusing [79], or by performing bulk operations on a remote processor. Instead of initiating remote operations one by one, the kernel combines all operations and initiates a single inter-processor request. In cases where the kernel has to repeatedly acquire locks, the fusing of multiple micro-operations can result in reduced concurrency, longer over-all waiting time for a lock, and potentially unfairness and starvation. Here, I explicitly move the policy decision via a hint to application level. Applications can then choose between coarse-grained locking with lower overhead or fine-grained locking but with a higher overhead.

Policy Freedom Of New Primitives

A primary design goal of L4 is to have no policies in the kernel. To a large extent that goal is achieved, with some exceptions such as the in-kernel scheduling policy. In this work I add new *kernel mechanisms* to achieve scalability. A primary design goal is to not introduce additional kernel policies.

Compatibility and Transparency

Developing an operating system from scratch is a tremendous multi-person-year effort. Therefore, I set a primary design goal that enabled me to leverage the numerous previous and ongoing system development projects for the L4 kernel [42, 54, 69, 72, 111]. I favored compatible kernel interfaces, mechanisms, and abstractions to the uniprocessor version over significantly different ones. For operations that violate isolation restrictions, I favor transparent completion with a performance penalty rather than explicit failure with notification. Transparent completion

preserves compatibility to existing uniprocessor software and algorithms, whereas explicit failure notification requires that applications be extended with multiprocessor error-handling code.

5.2.2 Multiprocessor Extensions to L4

Based on the design goals described in the previous section, I extend the uniprocessor L4 microkernel as follows:

Threads. The abstraction for execution is the thread. I extend the thread with an additional attribute that denotes a thread's *home processor*. The kernel does not automatically migrate threads, instead they are bound to their home processor. Thread migration (e.g., for load balancing purposes) is a user-level policy and only takes place upon explicit application request.

Address spaces. In L4, address spaces serve two primary purposes. First, an address space is a kernel object that associates virtual addresses to physical resources. Second, the address space provides protection boundaries. Threads which are associated with the same address space have the same rights to objects, including the right to manipulate peer threads in the same address space.

NUMA systems require processor-local memory (see previous section) and therefore different address translations depending on a thread's home processor. I separate virtual-memory address translation from thread permissions. A thread belongs to a permission domain that specifies the peer threads, and to a memory address space for the virtual-to-physical memory translations. An active thread can migrate between memory address spaces which provides a mechanism for implementing processor-local memory. The memory manager of the address spaces the thread migrates from and to needs to ensure that both address spaces have the same *memory content* mapped. The migration only changes the association to physical resources that is the memory access latency, but preserves the memory semantics (also see [109]).

Per-processor resources. Scalability requires parallel execution of independent application requests (see previous section). I replicate all central kernel data structures and provide per-processor instances. That includes *scheduling queues*, the *kernel memory pool*, and the *mapping database memory pool*.

Processor-local resources require an indirection that references the current processor's instance. I minimize the overhead induced by the indirection via the virtual memory subsystem. The per-processor kernel data structures are located in a specific area of the kernel's address space. Each processor has different memory mappings in that area, such that all accesses address the corresponding processor-specific instance of kernel data. On processors with software-loaded TLBs, the TLB-miss handler treats that area specially. On processors with hardware-walked page tables, the kernel creates

per-processor instances of the top-level page table. Each instance contains specific translations for its processor. For identical page mappings (such as user space and common kernel mappings) processors may share the same set of page tables.

One implication of the scheme is that data stored in the per-processor area cannot (easily) be accessed remotely, because it is only directly accessible by the specific processor.

Kernel messaging. I provide an in-kernel messaging mechanism for remote execution. The kernel uses a preallocated number of request buffers per processor to initiate a remote execution. The kernel sends an inter-processor interrupt in case the remote processor is actively running. If a processor runs out of free request buffers it busy-waits until a request buffer is freed up; the scheme avoids kernel deadlocks by continuously serving remote requests. All request handlers are constructed in such a way that the kernel can neither deadlock nor lifelock [117].

Read-copy-update epoch. For dynamic lock adaptation and also to avoid existence locks, the kernel uses read-copy-update epochs. At boot time the kernel forms the processor clusters (see Section 4.2.1). Based on the clusters it creates a ring of processors that circulate a token. The token is passed by writing a value into a dedicated per-processor memory field. Each processor regularly tests its associated field and when set, passes the token to its successor processor in the ring. The test for the token is at two locations in the kernel's code path: on user–kernel crossings and on timer ticks. The timer tick t_{tick} limits the maximum time for a round trip of a token t_{round} in a system with n processors to $t_{round} \leq 2n \cdot t_{tick}$.

User-kernel crossings include the performance-sensitive IPC primitive. The ring scheme generates two cache misses every time the token passes a processor. The first cache miss occurs when the token is read after it got modified by the predecessor in the processor ring. The second cache miss occurs when the token is passed on to the successor in the ring. Thus, in an n-processor system the scheme has an overhead of 2 cache misses every n system calls. Considering the high cache miss penalty relative to the overall cost of system calls, a strict token passing scheme incurs too much overhead, in particular when n is small. Based on the number of processors in the system, I reduce the cache-miss frequency by only testing every m^{th} system call. The required counter is local to each processor and thus remains in exclusive state in each processor's cache. Note that this optimization does not affect the maximum latency of a token round trip. In very large systems it is possible to form multiple sub-rings that include a subset of the processors in the system.

5.3 Inter-process Communication

Inter-process communication is the central mechanism for thread interaction, thread synchronization, controlled protection-domain crossing, and resource delegation. IPC performance is of paramount importance in a microkernel system. Operations which require two simple changes of privilege-level in a monolithic operating system require two IPC invocations in a microkernel. Thus, IPC adds an immediate overhead on all invoked system operations. Due to careful design, L4's IPC primitive has a very low overhead on uniprocessor systems. A detailed analysis of individual design decisions and principles was given by Liedtke [70]. In that paper, Liedtke briefly addresses multiprocessor systems and proposes a lock on the critical path. Such a locking scheme, however, would impose a performance penalty of up to 20 percent (see Section 6.2).

In this section, I describe the most performance-relevant usage scenario for dynamic lock adaptation for L4: the IPC primitive. Each thread in the system carries a processor isolation mask that denotes its primary scope for communication. According to the mask's restrictions, the kernel adjusts the lock primitive of the thread. Following, I will give an overview of the relevant communication scenarios in a microkernel-based system and their performance and latency requirements. Afterward, I discuss two design alternatives for a cross-processor IPC primitive. I describe the two alternative approaches to cross-processor IPC: (*i*) based on kernel locks and (*ii*) based on kernel messages. I conclude the section with a brief discussion on the IPC primitive for multiprocessor systems with significantly different cache migration overhead and IPI latencies.

5.3.1 Communication Scenarios

The microkernel's IPC primitive is used for a variety of purposes. But two scenarios are most common and relevant: (*i*) client-server interaction and (*ii*) IPC-based signaling and notification.

The following discussion is based on two core assumptions. Firstly, I assume that the programmer has detailed knowledge on application behavior and communication patterns of the workload. Design decisions always aim at maximizing performance for the *well-designed* application, rather than induce an overhead to accommodate unoptimized applications.

Secondly, based on the structure of current multiprocessor architectures, I make the assumption that the overhead for cache migration, synchronization, and signaling is so high that frequent cross-processor interaction in the sense of a remote-procedure call is unfeasible. This assumption stems from (*i*) the overhead for asynchronous inter-processor signaling, (*ii*) the cost for cache-line migrations (even in multi-core systems) relative to the frequency of remote invocations and the potentially achievable speedup. New features in upcoming processor architectures, however, may change the specific performance trade-offs. Furthermore, I do not address resource interference between threads of SMT processors, but consider

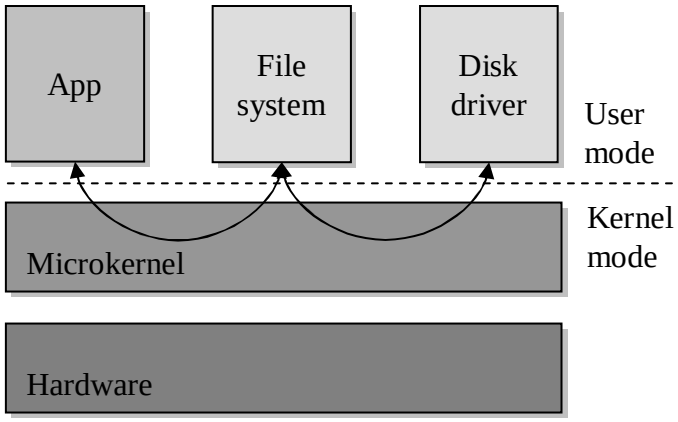


Figure 5.1: Client-server IPC model for OS services on a microkernel. Arrows denote the execution paths for IPC.

them as independently executing.

Client-Server Communication

In a microkernel-based system, the operating system functionality is not executed in privileged mode in the kernel, but unprivileged at application level. In order to invoke an operating system service, applications send an IPC to an OS server. Client-server communication is the most performance-critical operation, because it replaces all simple kernel-invocations in a monolithic OS by two IPCs.

In most cases, the client depends on the results of the server invocation and needs to block until completion of the remote operation. The IPC invocation thus follows the instruction stream of the application similar to a normal system-call invocation in a monolithic kernel. However, instead of stopping user-level activity and entering kernel mode to execute the system function, the kernel switches to the OS server's address space instead (see Figure 5.1). The switch is initiated by the client (by invoking the IPC primitive), authorized by the server (by accepting a message from the client), and executed by the microkernel.

IPC in the client-server scenario is therefore a mechanism to (*i*) guarantee that the remote server is in a consistent state (signaled by waiting for an incoming message) and (*ii*) to transfer control into the server's address space. It is important to note that client-server relations do not usually make use of the parallelism threading would provide, but *explicitly avoid* it. In order to minimize the IPC overhead, the scheduler is not invoked when switching from client to server thread and the server executes on the client's time slice instead. This scheme was first proposed by Bershad et al. [19].

In multiprocessor systems the client request could theoretically be handled by a server thread on a remote processor; the following reasons, however, make such an approach unfeasible for *blocking* requests:

- A client that relies on the result of the server request will be preempted, while the server needs to be activated. Performing such an operation across processor boundaries requires (*i*) invocation of the scheduler on the client's processor and (*ii*) activation of the server on the remote processor (which may further lead to preemption of an active thread on the remote CPU).

- The server requires access to client data in order to handle a client request (at least for the request parameters, but potentially more). For remote operations, such data have to be transferred via memory instead of the processor's register file. Hence, a remote invocation requires more instructions, and also incurs overhead due to cache line migrations. Furthermore, operating on memory that is shared between client and server incurs overhead for migration of the active cache working set between processors.[1]

- There is an inherently higher overhead for starting a remote operation compared to a local operation (even if small on tightly-coupled systems). The initiation of a remote operation either raises an interrupt or at least writes into a shared memory location (resulting in cache line migration). In the local case, the operation stays within the instruction stream of one processor.

  Only in one case a remote operation may yield *better* performance, that is if the overhead for context-switching from client to server is higher than the overhead for remote processor signaling plus the working set migration. Such scenario may occur, if the to-be-transferred cache working set is very small, the remote processor already activated the address space of the target server, the cache working set of the remote server is already loaded, and the processor is idle (i.e., does not require interruption).[2]

- Dynamic migration of worker threads is unfeasible on multiprocessors due to the cache migration overhead. Although, on well-designed SMT and tightly integrated multicore systems this requirement may be of less importance.

Based on the previous analysis I made the design decision to strictly favor the performance of processor-local IPC over cross-processor IPC. This decision imposes a design discipline that minimizes cross-processor communication and has general implications on the structure of the system.

Unrau's first design rule (preserving parallelism) requires that independent application requests can be served in parallel. When applied to a client-server operating system that rule requires *at least* one server handler thread per processor. In order to scale, the server needs to be aware of the parallelism and provide a sufficient number of worker threads. The threads have to be distributed across all

[1] SMT systems often share the L1 cache for different threads. Thus, the working set remains in the same cache if the processor supports L1 sharing.

[2] Each processor has a dedicated idle thread. Since it only executes in kernel mode there is no reason to switch to a special address space. Upon activation the idle thread therefore remains in the last active user address space.

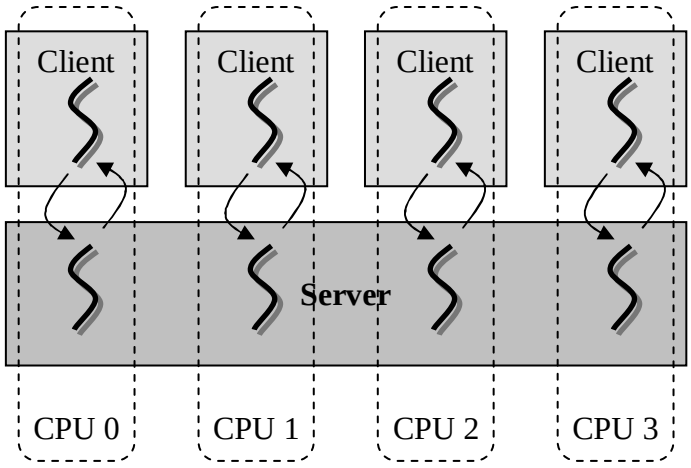


Figure 5.2: Example for client-server configuration with a multiprocessor-aware server. The server provides processor-local threads to eliminate cross-processor IPC. (The microkernel is omitted in the figure.)

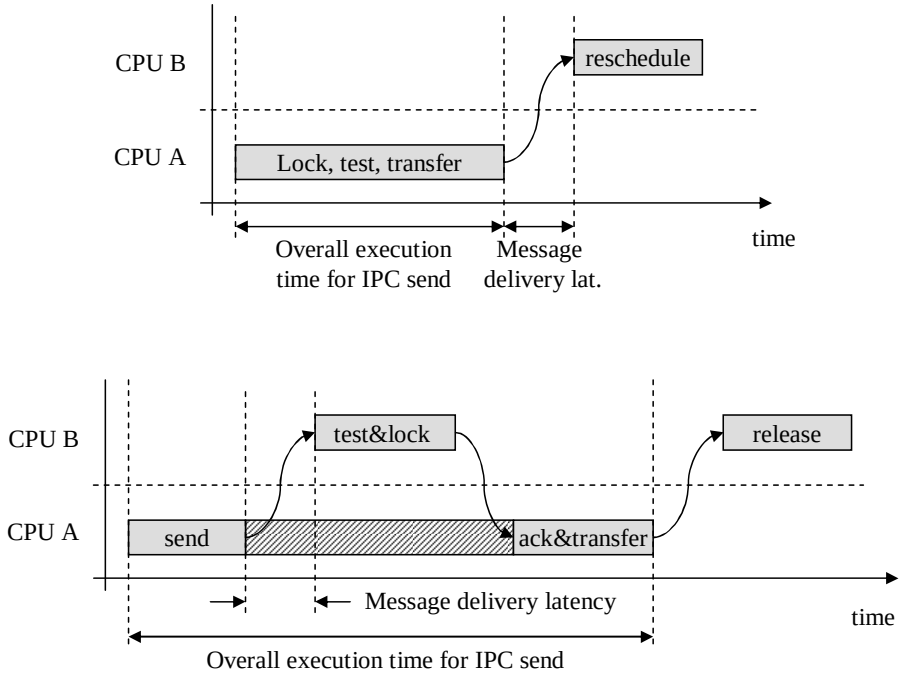
processors such that clients can contact a local server thread and thus avoid the described overhead for the remote invocation (see Figure 5.2).

Synchronization and Notification

Synchronization is a latency-sensitive operation. The time from the release operation of a synchronization primitive until a previously blocked thread starts executing influences the responsiveness and increases the overall runtime of a task. The design goal is to minimize that latency. The cost for an IPC that unblocks a thread on the same processor is low when comparing to alternative high-level kernel synchronization primitives (such as semaphores or mutexes). Hence, IPC is an sufficient mechanism for inter-thread synchronization. Furthermore, using one general mechanism for a variety of purposes is advantageous because it reduces kernel and application complexity, as well as overall cache footprint.

Signaling across processor boundaries requires three operations: (*i*) test and modify the thread's state from blocked to ready, (*ii*) enqueue the thread into its home-processor's scheduling list, and (*iii*) notify the remote processor to reschedule. These three steps can be realized in two different ways. Firstly, the data structures are manipulated directly and therefore require some form of synchronization (e.g., locks). Secondly, the operation is initiated via an in-kernel message and performed by the thread's home processor itself. In that case no locks are required because operations are serialized by the remote CPU.

Direct manipulation has a performance advantage because it allows a thread to perform multiple remote notifications without being dependent on the execu-



(b) Message-based IPC

Figure 5.3: Comparison of the message-based and the lock-based cross-processor IPC mechanism. Message-based IPC has a significantly higher latency.

tion of remote processors. It uses the parallelism of the memory subsystem that operates independently from the processors. An in-kernel messaging scheme can only achieve the same level of parallelism and overhead when the notification is asynchronous and without completion notification.

Such an approach moves the complexity of error handling to applications. The number of potential usage cases that do not require completion notification is quite limited. Moving this functionality to user-level results in significant code duplication because every user-level application would have to consider the case of lost messages. Furthermore, the common method for detection of lost messages is timeouts which should be avoided at all cost.

Waiting for completion of the remote operation can be achieved in two ways. Either the sending processor polls for message arrival on the remote processor (which adds the cross-processor signaling latency to the operation), or the initiating thread blocks until arrival of a confirmation message. The latter adds the overhead of a context switch and a scheduler invocation to each cross-processor notification. Figure 5.3 compares the alternatives and highlights the latency implications of each approach.

5.3.2 Performance Trade-offs

The design of the IPC mechanism on multiprocessor systems has to take a large number of parameters into account. These are not limited to performance and latency, but also user-level communication protocol complexity, transparent process migration for load distribution, atomicity and non-interruptibility of operations. A detailed reasoning is beyond the scope of this thesis and is discussed in [109].

Here, I only give the design parameters of the multiprocessor IPC primitive.

- Similar to the uniprocessor version, IPC addresses individual threads. Thread locality is explicitly exposed to application level. Multi-threaded servers need to expose the processor-specific thread IDs to communication partners (i.e., clients). This design decision favors IPC performance over locality transparency.
- For protocol transparency, the IPC primitive must work on the same processor as well as across processor boundaries. Limiting IPC to threads that are located on the same processor would require that all applications support error handling in case they are migrated to a different processor while in the middle of an IPC (e.g., waiting for a response of a thread that is suddenly on a different processor).
- IPC should be a generic, low overhead, and low latency primitive such that it is applicable to a wide range of scenarios. Multiprocessor support must have minimal impact on local communication.

As I argued in the previous section, cross-processor thread synchronization is extremely latency sensitive and an in-kernel message-based synchronization scheme performs less efficiently than a lock-based scheme. On the other hand, the majority of all IPCs take place between threads that reside on the same processor. Here, the locks required for efficient cross-processor IPC induce a constant overhead in the local case and a messaging scheme would be preferable for the few cases of thread migration.

When considering the most common communication relations of threads in a system, one notices very specific patterns. For each thread it is possible to derive a communication profile to partner threads that are either primarily on the same processor or primarily on remote processors. For example, server threads that have identical peers per processor communicate locally in almost all cases and only in a very few exceptional cases to remote threads (usually as a result of a thread migration). Threads that frequently synchronize with remote threads have a higher number of cross-processor communications relative to local IPC.

Fortunately, applications are aware of the specific communication patterns of their threads. For example, a server spawns parallel handler threads for processor-local communication, while a parallel program creates remote worker threads that synchronize frequently. That application-specific knowledge can be used beneficially to fine-tune the communication primitive.

5.3.3 Adaptive IPC Operation

Instead of favoring one IPC primitive over the other, the kernel supports both IPC variants — message-based and lock-based. Having two alternative code paths in the kernel only incurs a marginal overhead because (*i*) support for cross-processor IPC was one of the fundamental design requirements and (*ii*) to a large extent the code is identical for both alternatives.

Based on its communication profile, each thread can derive a set of *preferred remote processors* that are expressed in a processor isolation mask. Based on that mask, the kernel fine-tunes the communication primitive; processors that are specified in the mask use a lock-based IPC scheme, while all others use a message-based synchronization scheme for IPC. Depending on the *order* specified in the processor mask, the kernel may further fine-tune the lock primitive itself (i.e., use spin locks or MCS locks).

Accesses are restricted to a single processor if the isolation mask only specifies the home processor of a thread (or alternatively no processor). In that case the kernel *disables* the lock on the IPC path using the dynamic lock demotion scheme as described in Section 4.3.2. I want to emphasize that the processor isolation mask performance-tunes the IPC primitive but *does not* enforce communication restrictions. Hence, it is still possible to communicate with threads that are outside the bounds of the mask, however, the primitive uses the more expensive message-based code path.

Dynamic adaptation provides the optimal case for two scenarios: high-performance lock-free communication in the local case, and low-latency communication for the remote case. Adaptability eliminates the requirement for a special kernel synchronization primitive. Figure 5.4 shows the state diagram of L4's IPC primitive. I specifically marked the multiprocessor extensions and dynamic locks. The code path contains two locks that are acquired for the common client-server communication case: one for the send phase and a second for the receive path. A complete round-trip IPC therefore requires a total of four locks.

5.3.4 Remote Scheduling for Lock-based IPC

After completion of a remote IPC operation, the partner thread becomes runnable and has to be considered by the remote scheduler. If the priority of the currently active thread on the remote processor is lower than the priority of the thread activated by the IPC, the remote processor should invoke the scheduler. Rescheduling is achieved by sending an IPI to interrupt the current execution. Alternatively, if the currently executing thread has a higher priority than the activated one, the thread can simply be enqueued into the ready list and the normal scheduling mechanism will activate it at some later point in time. That case requires no IPI and avoids the accompanying overhead.

In order to scale, the kernel has per-processor structures for the scheduler. These structures include one ready list per scheduling priority and a wakeup list to

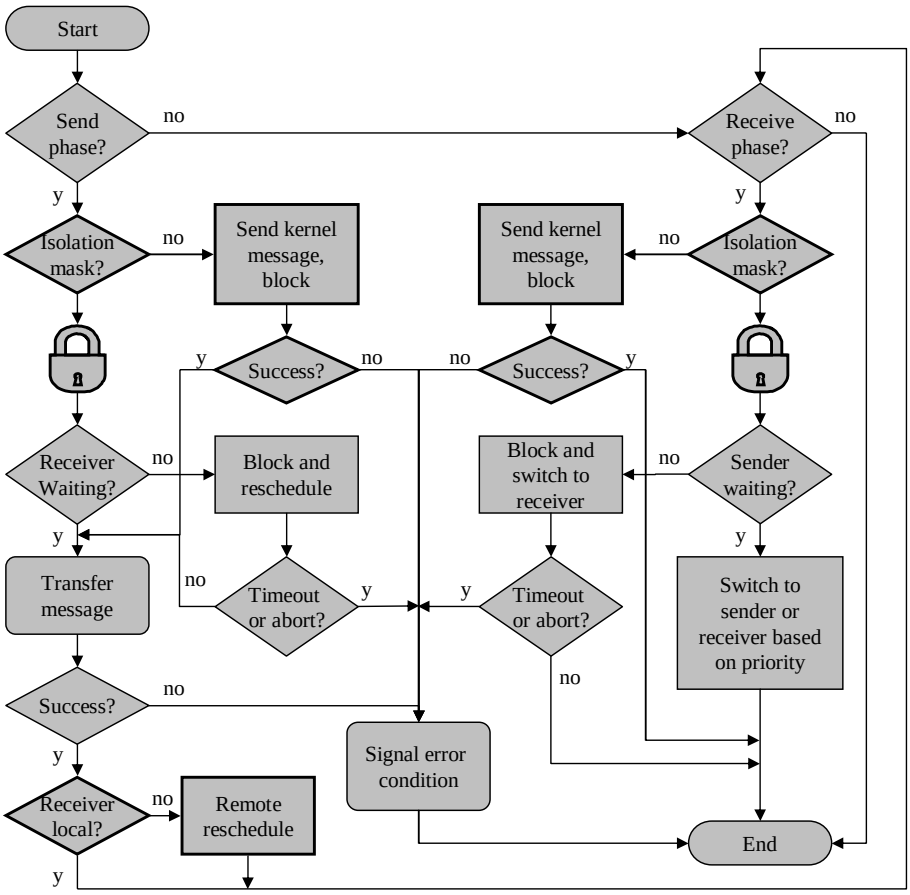


Figure 5.4: State diagram for multiprocessor IPC with support for dynamic locks and kernel messages. Elements with a bold border denote multiprocessor additions to the uniprocessor state diagram. The locks can be dynamically enabled and disabled. (Error handling is omitted for clarity.)

handle IPC timeouts. To lower the overhead for the processor-local case, scheduling lists are unsynchronized and therefore cannot be accessed from remote processors. Instead, each scheduler features one or more requeue lists that enable remote processors to use memory-based queues. When a thread is activated from remote, it is enqueued into a requeue list of its home processor's scheduler. Requeue lists are protected by per-list locks; multiple lists per scheduler further reduce concurrency from many remote processors and thus the potential for lock contention. Figure 5.5 depicts the structure for one processor in more detail.

Apparently, in order to support multiple requeue lists, each thread control block needs to provide multiple list pointers. However, that is not the case for the following reason: Threads only have to be enqueued into the ready list when their thread state switches from blocked to ready. That switch is a controlled operation via ei-

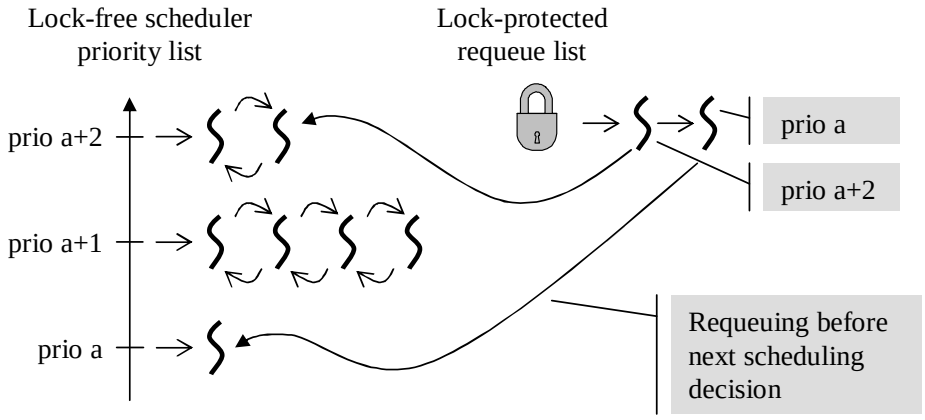


Figure 5.5: Requeue list for remote processors to avoid locks on the critical path for the scheduling queues. One list exists for each processor in the system.

ther IPC, a timeout, and a few other operations (see [109]). All state switches from a blocked to a ready state have to be synchronized via the IPC synchronization lock of the thread. When a processor holds that lock, no other processor in the system can concurrently manipulate the threads state; but then also no other processor will try to enqueue the thread into a requeue list and a single pointer is sufficient.

When the scheduler is invoked, it parses the requeue lists and requeues threads according to their priority into the normal scheduling queues. First, the scheduler checks the list head for valid entries and only then acquires the lock. Checking even a larger number of requeue lists therefore only incurs a modest runtime overhead.

The requeue list is a singly-linked list of thread control blocks; the complexity is $O(1)$ for insertions. Remote processors can only *add* members to the list and only the home processor can remove members as part of the requeue operation. Requeing threads therefore also has a complexity of $O(1)$ per list member, however, the latency for one specific thread with a total of n threads in the requeue list is $O(n)$.

5.3.5 Discussion on Tightly-coupled Systems

At the beginning of this section I narrowed the design space to systems that have noticeable cache-line migration overheads and relatively high IPI-signaling latencies. However, embedded multicore and SMT processors may show different behaviors. Those processors often have IPI latencies as low as ten cycles and efficient cache sharing with minimal cache migration overhead via on-chip direct cache-to-cache transfers. A thread allocation model with hard processor affinities per thread may be suboptimal. Following, I will discuss the implications of such system structure for the IPC primitive.

When the cache migration overhead between two processor threads reduces

significantly (or even drops to zero), a fixed thread association may lead to sub-optimal processor utilization. Multiple processor threads then better share one scheduling queue. Yet, to achieve maximum parallelism still requires one handler thread per processor thread in system servers. The direct IPC addressing scheme may lead to bottlenecks if all clients contact the same server thread. I envisage two alternative schemes to overcome the problem (without practically validating their effectiveness). First, server threads could be grouped such that multiple threads can be addressed with a single name. Alternatively, the kernel could append a processor tag to thread IDs of servers. Both schemes, however, introduce an in-kernel policy and the first scheme requires another lock on the critical path.

Specific lock primitives for such tightly coupled processors would provide a trade-off between shared data structures and high synchronization overhead (see also Section 7.2).

5.4 Address Space Management

In L4, address spaces and system memory resources are managed at application level. Address spaces are constructed by user-level servers. Initially, all physical memory is mapped to the root address space σ_0; new address spaces are constructed recursively by mapping regions of virtual memory from one address space to another.

The kernel's abstraction to memory is a *page mapping*. L4 lacks any knowledge on the *semantics* of the referenced page frames and treats all pages identically. For example, L4 does not differentiate between physical page frames that are RAM and others that are hardware devices. The advantage of such a scheme is a greatly simplified kernel abstraction with one uniform resource type. However, such uniformity may require overly conservative kernel algorithms with significantly higher overhead in runtime cost and memory footprint, or poor scalability.

I want to emphasize that the memory management mechanism in L4 only controls the permissions to memory resources. The kernel thus solely operates on *meta data*, such as page tables. In this section, I address the problem of significantly differing resource allocation patterns for memory resources in L4.

The section is structured as follows: First, I give a brief overview of memory management models in L4-based systems and describe their specific allocation patterns (refer to [12] for a more detailed description). Based on those patterns, I discuss the cost vs. scalability trade-offs and derive a set of requirements for the kernel mechanism. Then, I describe how these requirements are reflected in the kernel mechanisms that track the recursive memory mappings. A limiting factor for concurrency and thus scalability of the memory management mechanism is TLB coherency on multiprocessor systems. I apply the TLB versioning scheme described in Section 4.4 to the recursive address space model.

The recursive virtual address-space model enables user-level memory managers to perform NUMA-aware memory allocations, such as transparent page repli-

cation using processor-local memory. A detailed description is beyond the scope of this thesis and thus described in the accompanying technical report [109].

5.4.1 User-level Memory Management

User-level memory managers fall into two main categories: (*i*) managers that partition resources, and (*ii*) managers that operate on resources. The hierarchical memory model allows for re- and sub-partitioning via an intermediate address space. Partitioning is achieved by mapping pages into the intermediate address space, which then applies the partitioning policy and maps on the resources to the next level in the hierarchy. The relatively low kernel resource overhead for an address space makes such a simple model feasible. The second class of resource managers operate on the resources themselves. For example, a file system would allocate memory resources to cache files or a user-level server for anonymous memory would provide paged memory.

Memory allocations of applications also differ significantly depending on the applications' life span and level of parallelism. Applications allocate memory resources in three phases. In the first phase, the application populates the address space by faulting-in the active working set. Afterward, the application enters a steady state phase where it may access files or share data with other applications or servers. When the application finally exits, the kernel frees the address space and removes established mappings all at once. Obviously, for short-lived applications the set-up and tear-down phases dominate the resource management patterns, whereas for long-lived applications (and servers) the steady-state phase is more relevant.

While the specific resource management pattern depend on the specifics of application and memory resource manager, memory resource allocation follows three primary allocation pattern:

- frequent manipulation of very few pages (map and unmap),
- frequent manipulation of many pages at once for (*i*) address space tear down, and (*ii*) restriction of page permissions (e.g., for copy-on-write for UNIX's fork.), and
- infrequent manipulation of large sets of pages or super pages for bulk resource reallocation.

These allocation patterns are most critical for performance and scalability since they can result in long synchronization periods limiting scalability. Therefore, I will specifically focus the following discussion on them.

Rooting all physical memory from one address space (σ_0) is a very powerful and flexible mechanism, however, it also introduces a dependency chain between *all* address spaces in the system. Because intermediate memory managers (including σ_0) *could* modify or revoke mappings at any point in time, the kernel meta data

requires explicit synchronization. In the vast majority of cases, however, manipulations of the same permissions are restricted to a single processor. An overly fine-granular synchronization scheme incurs a significant runtime overhead and cache footprint. The overhead of a single taken lock is one to two orders of magnitude higher than the actual operation of updating the page table. Hence, an overly conservative synchronization scheme that solely focuses on scalability would induce a massive overhead.

Based on the common resource manipulation patterns I identify four scenarios that need special consideration for scalability as shown in Figure 5.6. The first scenario is where one page frame is mapped into two address spaces (Figure 5.6a). Subsequent mappings should be possible without interference and serialization of *B* and *C*.

The second scenario is the concurrent access or manipulation of mappings at different levels in the map hierarchy (Figure 5.6b). This scenario occurs (*i*) when the kernel restricts access rights and (*ii*) when the kernel reads access information to a page: the operation of the child *C* should be independent of other sub-mappings of parent *A*. Complete page revocation is less critical, because it eliminates the inter-dependency between parent and child.

The third scenario addresses mappings of differently-sized pages (Figure 5.6c). The parent *A* operates on a super-set of pages of the child *C*. Similar to the previous scenario, parent and child (and also siblings *B* and *C*) should be independent of each other.

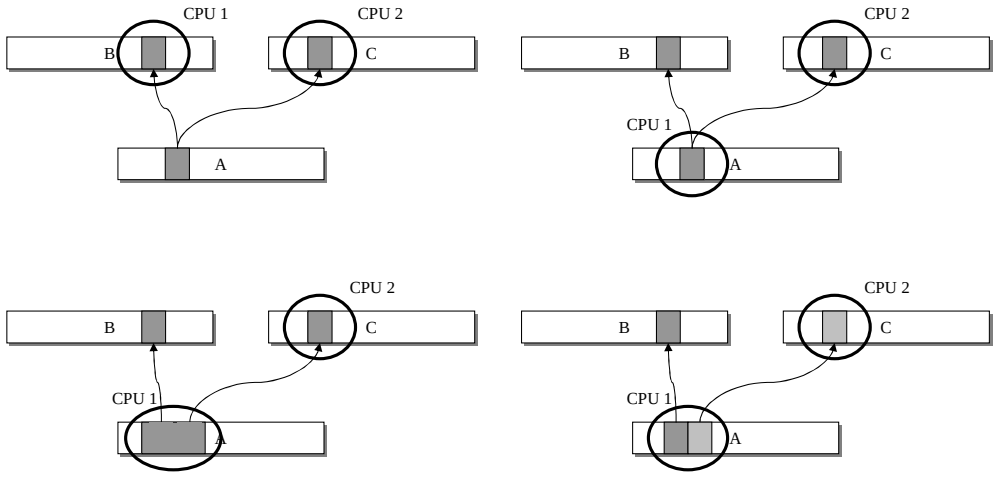
The fourth scenario is the concurrent manipulation of partially overlapping sets of mappings (Figure 5.6d). In the shown case, the parent manipulates mappings to *B* and *C*. Operations performed by the child in address space *C* should be independent of mappings to *B*.

A naïve solution that maximizes scalability for all four scenarios requires (*i*) complete independence of individual mappings and (*ii*) a TLB coherency scheme that is strictly enforced after *each individual permission update*. The latter requirement stems from L4's strict consistency model for page permissions. The invoked operation can only complete when permission updates are propagated to all processors and potentially stale TLB entries are updated (i.e., invalidated).

A simple cost analysis reveals that such a solution has a massive overhead that will eliminate and even reverse the benefits of improved scalability. For example, the overhead for a TLB coherency update is often multiple orders of magnitude higher than the cost for the actual in-kernel meta-data manipulation.

5.4.2 Scalability vs. Performance Trade-off

The scalability vs. performance trade-offs for the hierarchical memory management scheme are the common design trade-offs for all multiprocessor structures. Fine-granular locking results in better scalability while coarse-granular locking has a lower overhead. The memory subsystem additionally requires consideration of the overhead and latency for TLB coherency updates.



(c) Super pages and sub pages (d) Multiple combined pages

Figure 5.6: Scenarios of concurrent page permission manipulation in L4.

The scalability limitations for shared memory buses put a natural limit on the common degree of parallelism for memory pages. Common large-scale sharing of memory pages will result in poor performance due to the NUMA overhead. Instead, NUMA systems create replicas that are located in local memory with a lower access penalty. Hence, a well-designed system will only share very few pages across a large number of processors. Those shared pages are used for global data in massively parallel applications that run concurrently on a large number of processors. However, it is very unlikely that in those cases the page permissions change at a high frequency.

User-level memory management in a microkernel-based system is based on a mutual contract between two entities, one party that provides a resource and a second party that uses a resource. That contract includes access rights, availability, and for example guarantees on the content (e.g., that data is not arbitrarily modified or leaked). I add another dimension to that contract: *concurrency*.

Applications and system servers that implement the higher-level system abstractions have the required semantical information (which the microkernel lacks) for an educated decision on the synchronization strategy of individual memory mappings. Each partner in the resource contract has to express its preferable synchronization granularity and the expected level of parallelism. The resource contract is established when the resource permissions are transferred between both partners. The kernel correspondingly reflects that contract in its internal data structures.

Based on my initially stated scalability goals for L4, I support the following two scalability contracts:

- A new memory mapping can be created as dependent or independent from

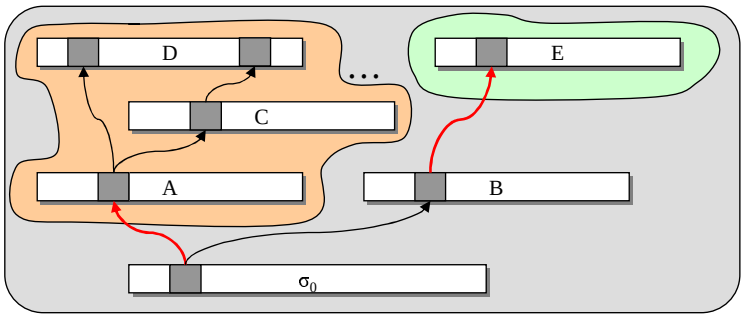


Figure 5.7: Memory subsystem boundaries; a boundary decouples the subsystem inter-dependencies.

the parent mapping. All dependent mappings are protected by the same lock while independent mappings can be manipulated concurrently.

- Sub-mappings of super pages can be synchronized either at the granularity of the super page or at the granularity of an individual mapping. The granularity can be dynamically readjusted by the owner of the super page at runtime.

These two potential contract types create subsystem boundaries that isolate subsystems and break the dependency chain to σ_0. Figure 5.7 illustrates this with a sample configuration.

The initially stated performance goal — minimal achievable overhead for an operation — focuses primarily on the overhead of the three common scenarios that I listed in the previous section. I specifically focus on the frequent (performance critical) operations, these are (*i*) frequent mapping and unmapping of only a few pages, and (*ii*) frequent manipulation of large sets of pages at once.

When manipulating the page permissions of a few pages only, fine granular locking has a relatively low overhead. This is in contrast to the cost of the overall operation, such as entering and exiting the kernel and potentially enforcing TLB coherency across processors. When manipulating the permissions of large sets of pages, the kernel should minimize the number of taken locks and even more important minimize the number of TLB shoot-downs. For such large-scale operations, I therefore use a delayed lock-release scheme. Before each release of a kernel lock, the algorithm tests whether or not the same lock would be reacquired immediately afterward. In that case, the algorithm avoids the overhead by skipping release and reacquire operations altogether.

5.4.3 Mapping Database

The kernel keeps track of recursively delegated resource permissions in a data structure called *mapping database*. The mapping database is L4's most complex data structure. For each physical page frame the kernel maintains an n-ary tree

that is rooted from σ_0. A variety of alternative data representations have been proposed before; each structure thereby addresses a different kernel design rationale. Völp [116] and Hohmuth [57] focus on avoiding long interrupt latencies and unbounded priority inversion. Haeberlen [51] proposes an encoding scheme that allows for partial preemption of kernel resources that back the mapping database structure. Szmajda [104] focuses primarily on optimizations for systems with software-loaded TLBs.

I extend L4Ka::Pistachio's uniprocessor implementation to accommodate the initially stated scalability goals and subsystem isolation requirements. First, I give a brief introduction on the general operations followed by a description of the multiprocessor extensions.

General Operations

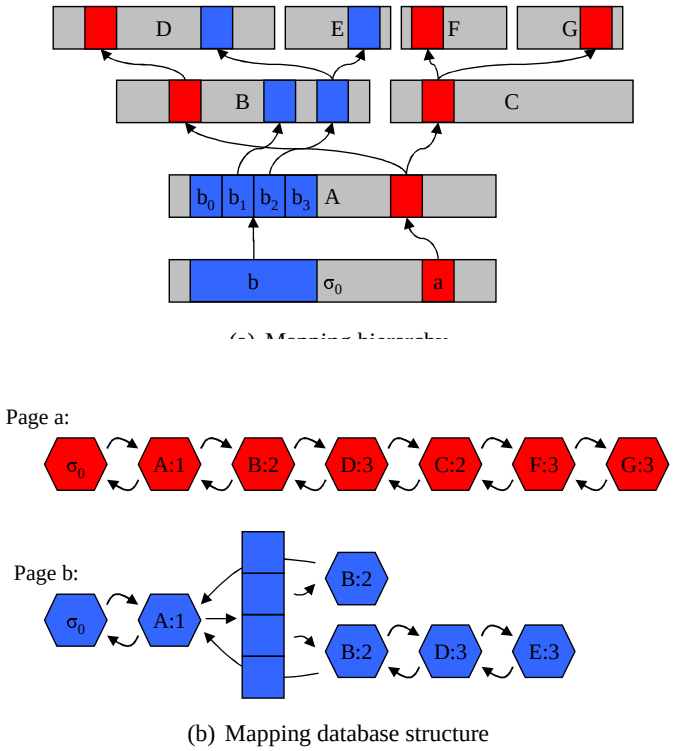
The mapping database has a number of design requirements. The database contains one entry per mapping that is stored in kernel memory, a scarce resource. Therefore, a compact data representation for mapping entries is of paramount importance. The restricted kernel stack space requires non-recursive algorithms. Insertions and deletions of entries need to be fast operations with a complexity of $O(1)$. Finally, a primary functional requirement is the support for different hardware page sizes.

L4Ka::Pistachio's mapping database realizes the n-ary tree via a sorted doubly-linked list of *map nodes*. The order of map nodes in the list reflects the map hierarchy. Sub-mappings are placed *after* their parent mappings and each node contains a *depth* field representing the level in the map hierarchy. When starting at an arbitrary map node in the list, all map nodes that follow directly and have a larger depth value are sub-mappings.

The mapping database is optimized for two performance critical operations: (*i*) finding the map node for a given virtual address (via its page table entry), and (*ii*) finding the page-table entry from a map node. For the first case, the kernel maintains a reference to the map node with the page table. For the second operation, the kernel stores a reference to the page-table entry in a compressed format within the map node.

The sorted list structure allows for an efficient insertion scheme. On map, the algorithm allocates a new map node and inserts it after the parent node. The node insertion into the doubly-linked list requires updating two pointers. On unmap, the algorithm walks the node list until it finds a map node that has a depth-field that is less or equal to the depth field of the start map node. The algorithm is sketched in the following listing.

```
1 StartDepth = Node.depth;
2 do {
3   Unmap(Node);
4   Node = next(Node);
5 } while (Node.depth > StartDepth)
```



(b) Mapping database structure

Figure 5.8: Mapping hierarchy and corresponding database structure for pages *a* and *b*. The level in the mapping hierarchy is represented by the numerical value. Page *b* is split into sub-pages via an indirection array. Capitalized letters denote address space IDs.

Super pages are realized via an intermediate array that has one start pointer for each sub page. Figure 5.8 illustrates a typical mapping scenario and shows the corresponding data structure of the mapping database.

Multiprocessor Extensions

The doubly-linked list is a space-conservative encoding, however, it requires the kernel to explicitly lock the list on insertions and deletions of nodes. The list based structure of the mapping database violates independence of subsystems and thereby contradicts one of Unrau's core requirements for scalability: preserving parallelism of independent applications. In the list structure, the last map node of one map tree and the top node of the next subtree cross-reference each other. Manipulation of entries in one subtree therefore requires locking of neighboring subtrees. Such structure either requires an overly broad locking scheme (e.g., on a per-subtree basis), or multiple locks for a simple list manipulation. Multiple locks additionally introduce the danger of deadlocks, which then has to be addressed via lock ordering or specific deadlock detection schemes. Although careful locking

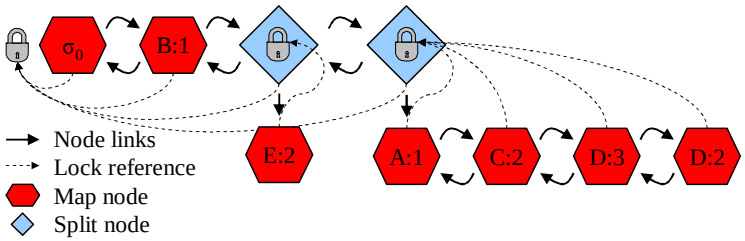


Figure 5.9: Mapping tree structure including split nodes. Concurrent operations are possible within independent subtrees.

can guarantee structural integrity of the data structure, the doubly-linked list will still require locks that cover independent nodes and subtrees. Because the selection of neighboring nodes depends on the order of previous map and unmap operations, applications can neither anticipate what other nodes an operation affects, nor what nodes an operation depends on.

I address this problem by dynamic structural modification of the n-ary tree. I introduce *split nodes* that structurally separate independent subtrees. The creation of split nodes is under control of both partnering applications at the time of mapping. Split nodes thereby serve two primary purposes. First, they structurally separate the mapping tree into multiple independent subtrees. Second, split nodes also serve as synchronization objects. Figure 5.9 depicts the kernel structure of the previous example extended with the additional subsystem boundaries.

Split nodes are inserted into the normal doubly-linked list and contain a reference to the independent (split off) subtree. The lock within the split node serializes accesses to all entries within one subtree, however, not across split-node boundaries (i.e., child split nodes). Each map node contains an additional reference to its parent split node, such that the corresponding split node can be found without requiring an intact list structure or a lock. The lock for a specific subtree can be derived by a double pointer indirection: the reference to the map node is stored with the page table. The map node itself then references the corresponding split node. The following listing shows a code sketch for a mapping database lookup including the split-node lock.

```
1 MapNode = LookupPageTable(vaddr);
2 if (MapNode != NULL) {
3   SplitNode = MapNode->SplitNode;
4   SplitNode->Lock();
5   if IsValid(MapNode) {
6     /* perform operation */
7   }
8   SplitNode->Unlock();
9 }
```

Super-page Locking

Applications can map sub-pages out of a larger super page. The mapping database reflects the division of a super page with an intermediate node array. The array has one entry for each sub page, and the specific number of entries is determined by the hardware-supported page sizes.

The mapping database is rooted from one top-level map node that covers σ_0's complete address space. That super-node is then divided via an intermediate array that covers a large set of super pages, which then gets further subdivided, and so on. The topmost map node has an associated split node that protects the complete mapping database with a giant lock.

When a memory manager splits a super page into multiple sub pages, the manager has two alternative locking strategies. First, it can apply a giant lock on all subtrees and thus minimize the lock overhead and lock footprint. Alternatively, it can lock individual sub pages with a higher overall cost but increased concurrency. The optimal locking strategy depends on a variety of parameters, such as the specific page frame (RAM vs. device memory), the frequency of remapping, level of sharing, and others. However, for resource managers that have long resource hold times (i.e., a root resource manager), the optimal locking strategy may change over time.

In absence of one general strategy, I provide dynamic adaptation of the lock granularity for sub pages that are mapped out of one super page. The map-node array is protected by a cascade of dynamic locks, using the scheme I developed in Section 4.3.3. In order to lock an entry within the array, the kernel first acquires a primary lock that protects the complete array, followed by another lock for each individual entry. However, the adaptive scheme effectively disables one of the locks — either the primary coarse grain lock or the fine grain locks. Switching between either lock granularity is under control of the memory manager that maps the super page. When establishing a new mapping, the memory manager can specify whether the locking strategy for the super page should change or remain as is. The memory that backs the fine-grain locks, is allocated and deallocated on demand.

5.4.4 TLB Coherency

L4's page permission model uses a strict consistency model. After completion of an *unmap* operation, the kernel guarantees that permissions are revoked. For the multiprocessor kernel I had two design choices: (*i*) restrict the consistency model to a single processor only and push the consistency to user level, and (*ii*) extend the strict consistency model to all processors.

The first model is unfeasible for the following reasons: User-level enforced consistency requires that a resource owner trusts the resource consumer to perform the TLB invalidation cycle on all processors. Alternatively, the resource owner would need one thread on each processor on which the page *may* be accessed. This requirement applies throughout the whole map hierarchy. L4's resource delegation

model is integral part of IPC. Restricting the extent of resource access permissions for processors via IPC restrictions is not only cumbersome but violates orthogonality of principles. I therefore chose the second model. Besides reduced complexity, in many cases it is possible to beneficially use architecture-specific behavior, such as IA-32's forced TLB invalidation on context switches [29].

I defined the following design goals for the mapping database's TLB coherency scheme that are in line with the overall scalability and performance goals:

- *Minimize number of remote TLB shoot-downs.* TLB shoot-downs have high startup and execution costs and the kernel should therefore try to avoid them.
- *Minimize number of affected processors per TLB shoot-down.* TLB shoot-downs should only target processors that actually require a shoot-down, instead of a global broadcast. This design goal is also a requirement for scalability, because otherwise the overhead for TLB shoot-downs would increase with the number of processors in the system (and thus violate Unrau's second design rule).
- *Minimize dependency between parallel operations.* The completion of an operation must be isolated to the specific resource and not depend on the completion of others.

In order to achieve these goals I introduced a number of kernel-internal tracking mechanisms. Each address space object carries an additional processor tracking mask as described in 4.2. The mask keeps track of potential pollution of TLBs and is derived from the locality of all threads of the address space. When threads migrate between processors the kernel accordingly updates the mask. The kernel derives the overall set of processors that require a TLB shoot-down, by merging the processor masks of modified address spaces upon permission revocation.[3]

For unmap operations that cover a large number of mappings, the kernel performs a single combined TLB shoot-down at the end of the update. This optimization, however, may lead to a consistency problem when multiple processors operate on the same mappings. The modification or removal of a map entry by one processor may make the other processor complete its operation *before* all TLBs are in a consistent state. This can happen if the state of the mapping database is already in the expected state. For example, two processors concurrently reduce page access permissions from read-write to read-only. The first processor updates all page permissions but postpones the TLB shoot-down. The second processor now also reduces the permissions, but only finds read-only permissions. No updates are required and therefore no TLB shoot-downs. However, only until after the first processor finishes its TLB coherency update is the operation safe. Hence, the initially described scenario would show incorrect behavior that could lead to data leakage or corruption.

[3] For a definition of the merge operation refer to Section 4.2.2.

There are two critical cases that need special considerations. In the first case the page permissions are reduced, however, the page itself remains mapped. In the second case, the page permissions are revoked completely. The problem can be further divided into two parts, an existence problem of map nodes and the actual TLB shoot-down. In the following I address both problems in detail.

Map-Node Existence

A thread that performs an unmap operation specifies an area of its own virtual address space. The kernel uses the thread's associated page table to find the corresponding mapping node and all subsequent mappings. Hence, when the unmap operation clears the page table entry, it also removes the evidence of the previous mapping. Any other thread incorrectly considers the virtual memory area as clean. In an alternative case, the thread still finds a reference to the map node, however, until it actually manages to lookup the entry it is removed by another processor.

I address both problems via a read-copy-update release scheme. Manipulations of map nodes are always protected by a lock on their superseding split nodes. When a map node is removed, the kernel leaves the reference to its split node alive, however, it marks the node as invalid. Invalidating the map nodes enables the algorithm to detect the described race condition.

In order to preserve references to completely unmapped nodes requires special handling in the mapping database. When the kernel revokes the page permissions in the page table, it marks the page table entry as invalid but leaves the reference to the mapping node intact. Subsequent map and unmap operations not only check for a valid page table entry but also for still active map nodes. The map node references serve as an indicator for potentially still ongoing parallel operations.

All freed map nodes are enqueued into an RCU free list with their split-node and page table reference pointers still intact. When the RCU epoch expires, the kernel finally clears the map-node references from the page tables before the map nodes are put back into the free memory pool. Clearing the map-node reference or overwriting it with a new mapping requires an atomic compare-exchange to detect potentially concurrent map attempts.[4]

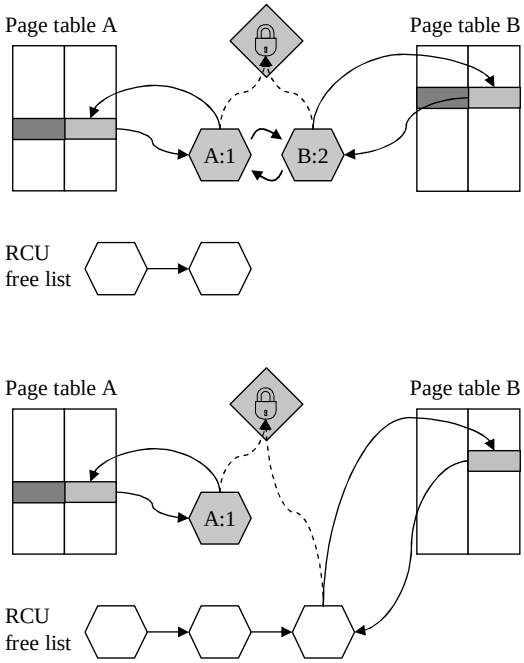
Figure 5.10 illustrates the described scenario for two address spaces.

TLB Shoot-down

The kernel synchronizes outstanding TLB coherency updates on split nodes. With the described map-node tracking scheme, the kernel can always safely derive the associated map node for a particular page mapping. The map node then references the corresponding split node.

TLB updates are parallelized with the TLB versioning algorithm I described in Section 4.4. The kernel allocates one TLB version vector per processor. The

[4] Unmap and over-mapping (one map entry is replaced by another) have a few corner cases that require special care. However, a detailed discussion is beyond the scope of this work.



(b) After unmap

Figure 5.10: Read-copy-update map-node release. A page is mapped in address spaces *A* and *B* and has corresponding map nodes. Both map nodes hold references to a common split node (a). After an unmap of the page in *B*, the map node is unlinked from the mapping tree and enqueued into the RCU stand-by list (b). The references between the page table, map node, and split node remain intact.

split nodes contain three fields for TLB tracking: (*i*) a cluster mask for tracking the affected processors, (*ii*) a TLB version vector epoch counter, and (*iii*) a reference to the TLB version vector. After a modification of page permissions but before the release of the split node lock, the kernel updates all three fields as follows:

When iterating over a mapping tree, the kernel computes a logical OR of all cluster masks of modified address spaces. The kernel then stores the computed cluster mask within the split node thereby denoting which processors require a TLB shoot-down in order to achieve a consistent state. Then, the kernel updates its TLB version vector and merges in the old version vector (see Section 4.4 for details). If there is an update in progress (i.e., the mapping tree was modified but the TLB shoot-down is not yet finished), the old and new cluster masks get merged. The TLB epoch counter stored with the split node enables the kernel to detect such an outstanding update. If the epoch counter is still identical to the processor's current epoch, the TLB shoot-down is not yet completed and requires special care. The kernel then updates the reference to the current processor's TLB version vector as well as the version vector epoch counter. After completion of all these updates,

the kernel releases the split node lock.

Map and unmap operations achieve a consistent state by performing the outstanding remote TLB shoot-downs themselves. Based on the processor's version vector and the calculated processor-cluster dirty bitfield, the kernel derives the remote processors that require a TLB shoot-down. However, instead of doing a brute-force shoot-down based on the dirty bitfield, the kernel checks the remote versions first and only forces invalidation for those processors, where the TLB version did not advance yet. Hence, if two processors operate on a set of pages and one operation completes earlier and triggers the TLB invalidation, the second operation may not even require another TLB invalidation. Furthermore, on architectures where the TLB is automatically invalidated on context switches, normal context switch activities may eliminate the necessity for a TLB shoot-down altogether.

5.5 User-level Policy Management

Policy-free kernel primitives is one of the primary design goals for L4, and also applies to the multiprocessor extensions. Per-processor resource allocation and scheduling requires explicit mechanisms that allow user-level servers to perform dynamic re-allocation and resource balancing. The two primary resources are threads (for load balancing) and per-processor kernel memory.

Efficient allocation of kernel resources from application level requires runtime feedback and safe mechanisms for resource re-allocation. In this section I describe the application of event logging and user-controlled resource management for individual processors in the multiprocessor configuration.

5.5.1 Kernel Event Logging

The information on resource utilization is distributed across a number of components, including the microkernel and also multiple application-level system servers. While in the majority of cases many servers can trivially provide usage data to a user-level scheduler (such as access information for certain resources via the IPC interface) some information is solely available in the kernel.

In order to expose microkernel-specific runtime information, I extend the kernel with the event logging mechanism as described in Section 4.5. The two central design goals are *scalability* and *minimal performance impact* for the overall system. A per-processor log buffer with unsynchronized access preserves independence between the system processors. For minimal overhead I instrument the kernel with a set of hand-optimized assembler macros. Furthermore, I use runtime code adaptation for the very few events that are on the critical path for IPC. In case event logs are disabled, the kernel can remove the logging code via binary rewriting [105].

The kernel events that are relevant to the scheduler are specific to the individual scheduling policies. As argued before, a one-fits-all solution is unfeasible because

of the significant overhead for cache footprint and execution time. For example, the overhead for a simple performance counter read (rdpmc) on a Pentium 4 accounts for 141 cycles and a time stamp counter read (rdtsc) costs 88 cycles. An unconditional execution on the critical IPC path would induce a 14 and 9 percent overhead (or 23 percent when logging both counters).

The kernel instrumentation consists of three components: the event logging instrumentation for kernel events, a set of log control registers, and the log buffer. Log control registers and log buffers are located in user-readable memory, such that a user-level scheduler has fast access to the logged data.[5] Access to the log can be restricted to avoid leakage of security-relevant information.

The kernel uses a double indirection for logging. Each kernel event is associated with a specific control register. The control register contains a reference to the event log, a counter, and a start value for the counter on overflow. On every event the kernel decrements the counter; when it reaches zero the kernel logs an event. Afterward, the counter gets reinitialized with the start value. The log buffer is described by a 64-bit data structure that contains a reference to the buffer, the buffer size (encoded in a mask), and flags that specify the log format. Schedulers have the following per-log configuration flags:

- *Current principal.* Logs the current resource principal.

- *Event parameter.* Events have an associated event parameter that is specific to the event type. When enabled, the kernel writes the parameter into the log.

- *Time stamp.* The kernel supports four alternative time-stamp modes: (*i*) no time stamp (i.e., off), (*ii*) hardware time stamp counter (rdtsc on IA32), (*iii*) the kernel-internal time (via a periodic interval timer), and (*iv*) a per-processor counter, that is incremented on each log event (for a causal order of events).

- *Event counter.* The counter contains the delta of entry–exit events or the number of occurrences of a counted event. A separate flag denotes whether the new value overwrites the previous entry in the log or gets added. The special case enables a scheduler to either accumulate events or log the last event.

- *Event type.* Each kernel event has a unique event ID. By logging the event ID, it is possible to merge the events from multiple event sources into one combined log buffer.

[5] The system uses kernel-provided memory that is allocated at system start time. Dynamic kernel memory allocation schemes, as proposed by Haeberlen [52], are a viable alternative to avoid the limitations of static allocation. However, safe kernel resource management was of less importance for this work and is not further discussed.

The kernel provides two configuration interfaces, one for the event control (event counters and the associated log) and a second for the log itself. The association between kernel event sources and logs can be freely configured by the scheduler. Hence, it is possible to combine multiple events in a single log and also distribute the events into individual and independent logs. The log memory itself is a linear unstructured memory buffer and is managed by privileged applications. The kernel does not apply any consistency checks (besides simple boundary checks) and incorrect configuration will lead to log corruption, however, without endangering the kernel's consistency.

Event Sources

The most important kernel events are those that either cannot be derived without specific kernel support or require trustworthy applications. The latter is an assumption that does not hold for most scenarios. The kernel exports the following *per-processor* event sources:

1. *Change of number of runnable threads.* L4's in-kernel scheduler hides the number of runnable threads in the system. Previous research [14, 96] identifies the run-queue length as an important indicator for load imbalances and balancing decisions. This event source exposes the run-queue length of the current processor.

   Note that the common case of client-server IPC does *not* change the number of runnable threads in the system. Hence, the event is off of the critical IPC path.

2. *Idle time.* When a processor falls idle the kernel enters a low power state for that processor. The processor gets reactivated by a hardware, timer, or interprocessor interrupt. The overall per-processor idle time is relevant for a scheduler in order to re-allocate the load from highly loaded processors to underutilized processors. Idle-time logging is obviously less performance critical since the processor is idle anyway.

3. *Kernel memory pool.* The kernel maintains a per-processor memory pool that serves the main kernel memory consumers: thread control blocks, address space objects, page tables, and the mapping database. In order to scale, each processor has an individual memory pool. In case a pool falls short, memory has to be reallocated between processors.

   In order to keep the kernel policy-free (as stated in the initial design goals), pool re-balancing is a policy implemented at application level. The kernel mechanism is detailed in Section 5.5.3. The in-kernel memory allocator keeps track of the available memory in the free pool. Memory allocations and deallocations create log entries such that the memory manager is aware of each processor's current pool status.

Additionally to the per-processor events, the kernel has a number of per resource-principal events. L4's resource principals are threads and the most important system event is IPC. Previous research identifies communication patterns as an important indicator for process inter-relations. Scheduling policies consider those relations for more efficient thread placement. However, logging IPC on a per-thread basis is unfeasible due to the extremely high frequency of IPC in a component-based system and the large number of threads. On the other hand, many communication relations between threads are well-known, static, and irrelevant for a scheduler.

Based on the fundamental idea of resource containers, I introduce an additional logical accounting level on top of the kernel's resource principals: *thread domains*. Each thread is associated with exactly one domain and its association is managed by a privileged user-level server. Only switches between *different* thread domains trigger a kernel event. The domain ID is stored within the kernel's thread control block. For threads within the same domain the additional event logging runtime overhead is reduced to a simple comparison and a conditional, non-taken jump.

Domains have a configuration register that associates a log file to the domain. The association can be changed from application level and multiple domains may share a single log buffer. Furthermore, the kernel has one special domain — domain zero — that does not lead to event recording. Initially, all threads are allocated to that domain. By co-locating the domain ID with other frequently read data on the critical path, the execution overhead is negligible.

5.5.2 Thread Scheduling (Load Balancing)

L4 threads are bound to their specific home processor and the kernel does not apply a thread migration policy (a design decision detailed in Section 5.2.2). Thread migration is solely initiated by user-level load balancers. The core migration operation is *migrate thread T to processor P_{dest}*.

The frequency of thread migrations is bounded by the cost for migrating a thread's active cache working set. The primary thread migration scenarios are *initial placement* and *load balancing*. The kernel places newly created threads on the home processor of the creator; thus, thread creation is a processor-local operation that does not require interaction or synchronization with other processors. Thread creation is a privileged operation and can only be executed by a restricted set of (privileged) threads [49]. The design scales because the privileged server can spawn multiple, processor-local worker threads that can operate in parallel.

Thread migration is initiated via a system call that initiates thread relocation to the new home processor. Migration may be initiated from any of the processors, independent of the current home location of the to-be-migrated thread. Such flexibility is required to support a wide variety of load balancing and placement schemes such as work stealing [21] and push-based policies. There are three potential migration scenarios: (*i*) from the current to a remote processor, (*ii*) from a remote to the current processor, and (*iii*) between two remote processors.

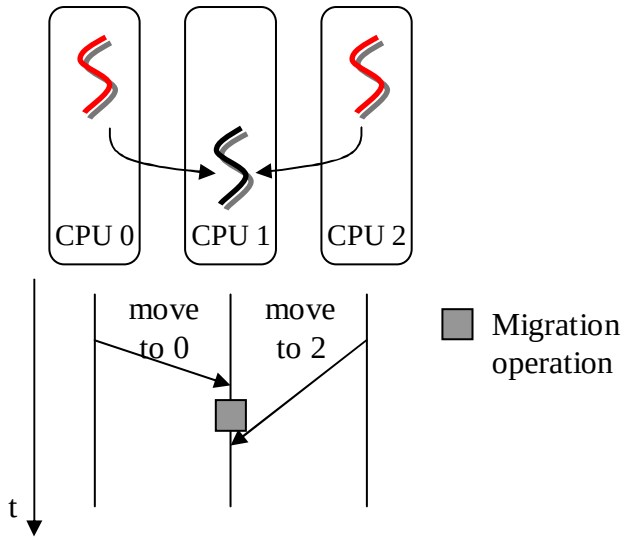


Figure 5.11: Concurrent migration attempts from CPU 0 and CPU 2. The thread on CPU 0 successfully completes the migration before the second migration request is started.

The lock-free per-processor scheduler structures force an in-kernel message-based scheme for migration requests. Because remote queues must not be accessed, a scheduler needs to send a request message to the migrated thread's home processor in order to dequeue it from all scheduling queues, update the home processor, and potentially update the thread's processor isolation mask. The migration is finalized by enqueuing the thread into the new home processor's requeue list. With the next scheduling interval the thread gets integrated into the normal scheduling queue and the migration completes.

Since thread migration can be initiated from all processors, the design may lead to a race condition. When two processors concurrently try to migrate the same thread, then both processors will initiate a migration request to the home processor. Global ordering of message-based kernel synchronization guarantees that one of the migration requests completes successfully while the other one fails (see Figure 5.11). The request that comes second will find that the to-be-migrated thread is not located on the home processor anymore. However, simplistic forwarding of the migration request to the thread's new home processor may result in starvation, since the thread may be already migrated.

I eliminate the race condition via kernel-user interface design. The solution is generically applicable to other user–kernel race scenarios and here exemplified for the thread migration case.

Concurrent migration requests (as described in the scenario above) represent an unsynchronized resource allocation scheme. Enforcing a strict order of operations

in the kernel requires additional synchronization. Such synchronization, however, moves synchronization from application level to the kernel and contradicts the initial design constraint: maximizing performance for the well designed application. Careful interface design can achieve the same result without the additional overhead.

Thread scheduling is based on the *current system state* and an *intended system state*. Thread migration is the operation that initiates the transition between both states. Two concurrent migration requests express one of the following four cases: (*i*) the first migration request did not complete yet and a new migration request is already initiated, (*ii*) the second migration request started after the first but arrived earlier due to message processing delays, (*iii*) the user-level scheduler did not synchronize migration at all (which effectively is a bug), and (*iv*) two schedulers run an attack against the kernel. Only the first and second case require correct handling by the kernel. In both cases the race condition manifests a timing problem of concurrently issued migration requests on different processors.

I enable applications to detect the race via an additional temporal parameter and thereby move the request ordering out of the kernel to the application. For thread migrations, the scheduler has to specify an additional parameter that identifies the *expected* system state (i.e., the scheduler's view on the current system state) expressed by the current home processor of the thread. Hence, a thread migration operation has three parameters: the thread T, the source processor P_{source}, and the destination processor P_{dest}. A race condition occurres if $P_{\text{source}} \neq T_{\text{home}}$ and the kernel operation fails completely. In that case, the scheduler can recover gracefully and restart the operation.

The kernel supports migration of groups of threads, which minimizes the per-thread migration overhead and also preserves scheduling inter-dependencies.[6] The operation is still safe and bounded by requiring that all threads have to reside on the same source processor and are migrated to the same destination processor. Since thread migration requires execution on the threads' home processor (which is identical for all) the migration is race free once started and guaranteed to either complete successfully or fail.

5.5.3 Kernel Memory Management

The recursive virtual address space model is a flexible mechanism that decouples memory management and memory allocation policies from the kernel. Yet, the kernel consumes memory for kernel meta data, such as thread control blocks, page tables, and the mapping database. In order to scale, each processor has a separate in-kernel memory pool.

Since control over the physical memory resources is at application level, the kernel depends on user-level applications to manage the kernel pools. Similar to

[6] Some synchronization constructs assume, that high priority threads always preempt low priority threads. When threads are migrated one-by-one, threads that have been mutually exclusive may run concurrently during the migration phase.

the solution proposed by Liedtke et al. [74], I use a cooperative memory resource allocation scheme between the kernel and a user-level memory manager. The kernel has one system thread per processor that handles memory allocation and deallocations. The root memory server, σ_0, can grant memory to the system threads which then place the memory into the per-processor pool. The operation is safe, because the kernel enforces that no user application has access rights to the page.

Per-processor pools incur a balancing and locality problem. The memory pool for allocation and deallocation depends on the resource principal (i.e., the thread) that performs the respective operation. Depending on thread locality and system structure, kernel memory may only be allocated on one processor and only released on another. Hence, the memory pool of one processor depletes while the other processor's pool is overfull. Furthermore, to preserve memory locality in NUMA systems, memory has to be local to individual processors. Mixing memory from different memory pools may result in poor performance when using remote memory for critical kernel meta data (such as thread control blocks).

I address both problems via strictly preserving pool locality for memory. Kernel memory is allocated at page granularity and free pages are maintained in a linked list. In addition to the primary free list, the kernel maintains two further memory lists, a *remote standby list* and a *requeue list*. When a page is granted to the kernel, the kernel records a *home* processor for the page in the mapping database. On page deallocation the kernel validates whether or not the page is freed on the home processor and either enqueues the page back into the normal memory pool or in the remote standby list. When the RCU token arrives at the processor, the kernel walks the standby list and relocates the pages from the standby list into the requeue lists of the pages' home processors. Furthermore, the processor places all pages from its own requeue list back into the free pool. Figure 5.12 illustrates the scheme for a configuration with three processors.

On processors that share one memory bus I enable an allocation optimization. Instead of assigning memory pages to one individual home processor, pages are assigned to a home processor cluster. The kernel directly places a page into the free list if it is released by *any* processor specified in the processor cluster. That optimization still preserves NUMA locality but reduces some overhead for requeuing.

Each processor logs the free pages in the kernel-provided event log. The user-level memory managers monitor the event log and rebalance memory between the memory pools. Rebalancing is realized by first revoking memory from the kernel and granting it back to the kernel on another processor (as shown in Figure 5.13).

5.6 Summary

In this section I described the application of dynamic locks, TLB coherency tracking, and event logging to L4Ka::Pistachio, an L4 microkernel. The fundamental design paradigms of L4 are minimalism, strict orthogonality of abstractions, and

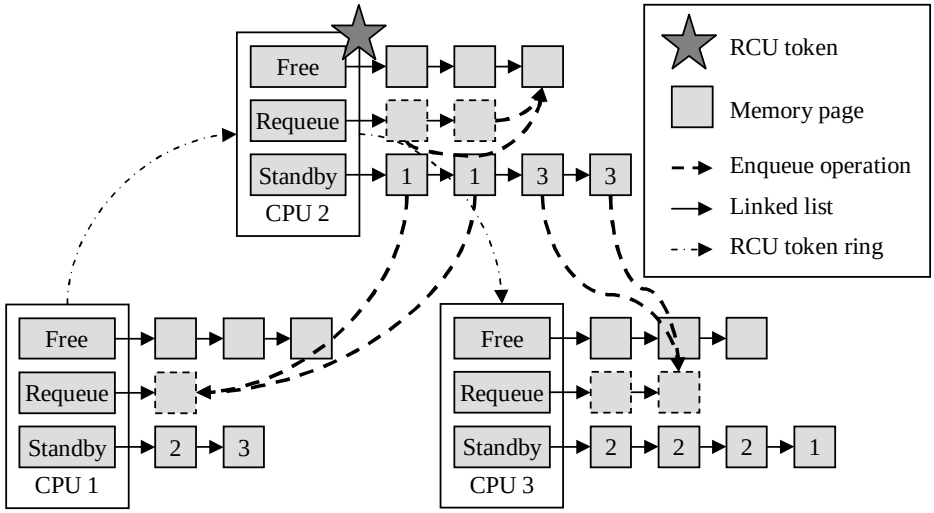


Figure 5.12: Per-processor kernel memory pools with processor local memory. CPU 2 holds the RCU token and moves the pages from the requeue list into the free list. Furthermore, CPU 2 empties its standby list and places the pages into the page owner's requeue list. Requeue operations require the possession of the RCU token. Since there is a single token in the system, no concurrent requeue operations can take place and list synchronization is not required.

user-level policies.

In microkernel-based systems, operating system invocations are replaced by inter-process communications. Hence, the performance of the IPC primitive is of paramount importance for the overall system performance. The additional overhead of multiprocessor synchronization primitives induces unnecessary overhead on the IPC mechanism. A second communication primitive that is specifically tailored for cross-processor communication would increase the kernel's cache footprint and also lead to higher application complexity. The cluster mask combined with dynamic locks provide a flexible and efficient solution to accommodate both communication scenarios with a single kernel IPC primitive (Section 5.3).

Section 5.4 described the multiprocessor extensions to L4's virtual address space model. L4 maintains a kernel data structure that tracks memory permissions of the address spaces on a per-page basis. I extended the kernel data structures in a way that applications can adapt data structures for coarse-grain or fine-grain locking. Furthermore, the developed tracking scheme for TLB coherency updates decouples parallel page permission updates.

Finally, in Section 5.5 I presented the application of the event logging scheme for user-level resource scheduling. In monolithic kernels, resource balancing policies are implemented as a kernel policy. A kernel policy contradicts L4's design principles. The presented event-logging mechanism provides runtime feedback

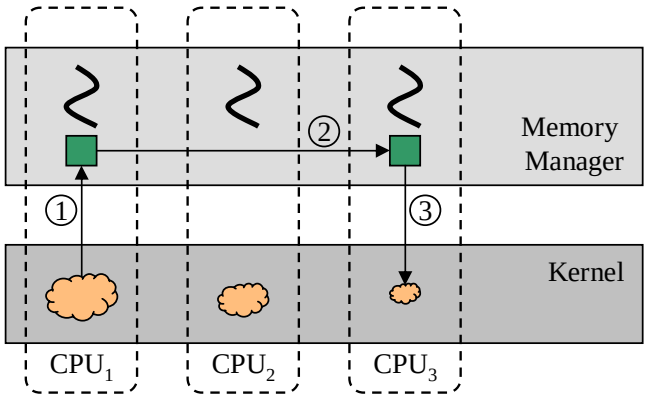


Figure 5.13: Memory balancing between kernel memory pools. In (1) the page is reclaimed on CPU 1, in (2) the managing thread on CPU 3 is notified, which then (3) places it into the local memory pool on CPU 3.

to a user-level load balancer which then can reallocate resources as needed. The scheme was used for two important kernel resources: threads and kernel memory.

The following chapter evaluates performance and scalability of the described design for the L4Ka::Pistachio kernel.

Chapter 6

Experimental Verification and Evaluation

In this chapter, I evaluate the performance of my kernel design implemented in L4Ka::Pistachio. I then evaluate the overhead and scalability of all individual kernel operations in a set of microbenchmarks. I compare the costs of individual kernel operations for different multiprocessor workload configurations, intra- vs. inter-processor operations and low vs. high concurrency. The scalability is evaluated according to Unrau's design requirements for scalable operating systems (preserved parallelism of operations, bounded operations independent of the number of processors, and memory locality). Furthermore, the benchmarks evaluate the *overhead* of multiprocessor primitives to the baseline performance given by the uniprocessor kernel.

Section 6.1 describes the hardware details of the evaluation platform that were used for the different benchmarks. Section 6.2 discusses the performance and scalability of the IPC primitive. Section 6.3 then evaluates the performance and scalability of the event-logging facility, followed by an evaluation of the kernel memory management subsystem in Section 6.4. Finally, I summarize the evaluation in Section 6.5.

6.1 Evaluation Platform

All performance evaluations were performed on IA-32 systems. The main test system was an IBM xSeries 445 server with eight 2.2 GHz Intel Xeon processors based on the Pentium 4 (P4) micro-architecture. Each physical processor had an 8KB L1 data cache, a 12KB μop trace cache, a shared 512 KB L2 cache, and a shared 2 MB L3 cache. It further featured a 64 entry data TLB and a 64 entry instruction TLB.[1] Each processor had two logical threads (HyperThreads) enabled. The system consisted of two processor boards with four processors each. Each

[1] Intel family 15, model 2, stepping 6

Operation	Overhead (cycles)
atomic exchange (xchg)	125
atomic compare-exchange (lock cmpxchg)	136
atomic decrement (lock dec)	123
address space switch (mov %r, CR3)	827
entering and exiting kernel (sysenter, sysexit)	135
entering and exiting kernel (int, iret)	839
L1 hit	2
L1 cache miss, L2 hit	19
L2 cache miss, L3 hit	43
memory access latency	206

Table 6.1: Individual overhead for operations for the test system.

board had separate memory (2 GB each) and both boards were interconnected via a proprietary NUMA memory interconnect developed by IBM.

The Pentium 4's micro-architecture has some specifics that I want to highlight. The P4 features a *trace cache* that caches instruction as translated μops instead of the normal instructions as read from memory. The cache is virtually tagged and the P4 does not support address space identifiers. On an address space switch the processor not only invalidates its 64 I-TLB and 64 D-TLB entries, but also flushes the trace cache and L1 D-cache. Hence, on every address space switch the processor invalidates a significant amount of its active working set resulting in a higher context switch overhead than other IA-32 micro-architectures, such as Intel's Pentium III and AMD's Opteron.

IA-32 has a relatively strong memory ordering model, defined as *write ordered with store-buffer forwarding* [29]. The highlights include: (1) reads can be carried out speculatively and in any order, (2) reads can pass buffered writes, but the processor is self-consistent, and (3) writes to memory are always carried out in program order (with some exceptions). Such a strong ordering model has implications on speculative execution and creates a high overhead on atomic instructions because the processor has to honor the ordering model and looses some optimization potential via speculative execution.

Table 6.1 lists the measured costs for a set of important operations of the test system including the costs for atomic memory operations, the costs for entering and exiting the kernel, and the costs for switching address spaces.

The test system had eight processors with a total of sixteen processor contexts. Literature suggests that scalability problems in many cases only become visible for systems of more than 32 processors. I was limited by the hardware constraints, however, I argue that the results are still relevant and expressive. As argued in 3.1.1,

scalability of operating systems is *not* evaluated by the speedup of an operation but by its response time. An operating system is considered scalable if the response time is independent of the number of processors in the system. In the following, I evaluate the overhead for individual kernel operations and their response time with increasing parallelism. I assumed that the hardware architecture of the test system sufficiently scaled.

The number of processors also limited the evaluation of the effectiveness of the cluster mask. While the reference implementation was able to handle the full cluster mask, the processors of the test system always fitted into the provided 16 bits of the bitmap.

6.2 Inter-process Communication

I evaluated the overhead of the IPC primitive for the most important communication scenarios: client-server communication and cross-processor communication. In the client-server scenario, I further differentiated between two cases: (1) inter-address space where both communicating partners resided in different address spaces and (2) intra-address space where both communicating threads resided in the same address space. The intra-address space IPC is relevant for worker-thread scenarios where a central distributor hands off requests to the worker but also for in-address space synchronization threads or memory pagers [12]. All microbenchmarks measured the overhead of the kernel operation by sending messages in a tight loop that was executed one thousand times. Afterward, I calculated the average cost for one IPC operation.

6.2.1 Processor-local IPC

Inter-address-space IPC is the common case for client-server interaction and is used to transfer request and response payloads for user-level IPC protocols; the operation is completely executed on one processor. The cost is dominated by the previously mentioned overhead due to TLB and cache invalidations.

I used the uniprocessor IPC primitive as the base-line performance and compared it against the multiprocessor IPC with disabled and enabled locks. The results showed that the performance for the lock-free version on multiprocessor systems was almost identical to the uniprocessor variant. The overhead accounted for a few additional checks that were required to differentiate between local and remote threads. These extra instructions increased the overhead by 30 additional cycles (or about 3 percent).

The overhead for IPC on the P4 microarchitecture was dominated by the cost of kernel entry and exit and the overhead due to cache and TLB invalidations on each context switch. The primary costs were due to entering and exiting the kernel (135 cycles) and the overhead for switching address spaces by reloading CR3. The cost of an address space switch including its follow-on cost accounted for 629 cycles.

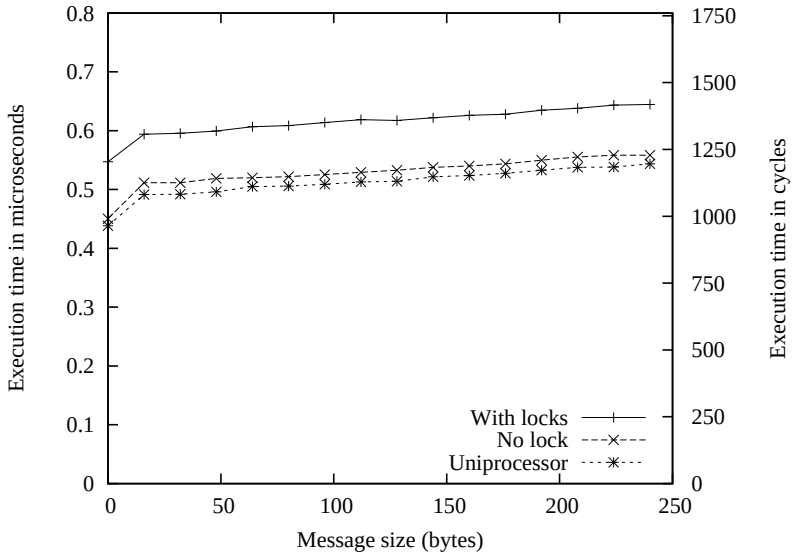


Figure 6.1: Inter-address-space IPC performance on the same processor for uniprocessor and multiprocessor with enabled and disabled lock primitives. The overhead is dominated by TLB and L1 cache flushes on the Pentium 4 architecture. The initial jump in the curve reflects startup costs for the message copy loop.

The remaining costs for the IPC and the benchmark code is 226 cycles. A more detailed break-down was not possible because of secondary effects in the microarchitecture. In some cases, the insertion of additional no-ops lead to performance improvements.

The IPC variant that had spin locks enabled showed an overhead between 16 to 20 percent, depending on the message size. The two required locks induced an overhead of 190 cycles, which was below the predicted 250 cycles. I partially attributed the difference to additional wrapper code and partially to secondary effects such as pipeline stalls and serialization (more details follow).

Figure 6.1 shows a detailed graph that compares IPC on a uniprocessor kernel against IPC on a multiprocessor kernel. The graph shows the performance of both multiprocessor IPC variants with enabled locks and disabled locks.

Figure 6.2 shows the same microbenchmark as before, except for intra-address-space IPC. The overhead between the uniprocessor and the multiprocessor variant with disabled locks was between 30 to 80 cycles depending on the message length. However, the cost for uniprocessor IPC did not show the linear increase of runtime relative to message length and even dropped significantly for longer messages. I was able to confirm the effect in multiple experiments and I attributed it to specifics of the P4 microarchitecture. The copy loop used an optimized copy mechanism that favors complete cache line transfers. There were no serializing instructions on the critical path allowing the processor to perform heavy speculation. The curve showed the expected constant increase with introduction of the additional multi-

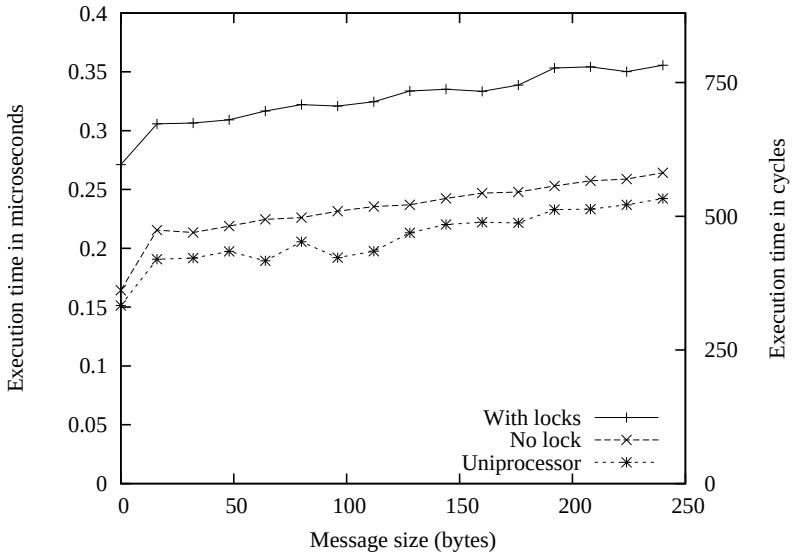


Figure 6.2: Intra-address-space IPC performance on same processor with enabled and disabled lock primitives. The cost of the lock primitive is dominant when compared to inter-address-space IPC where the highest overhead is due to TLB and cache flush.

processor support code. The lower overall cost of intra-address-space IPC resulted in a higher relative increase due to spin locks. The two additional locks added between 35 to 64 percent overhead to each IPC.

6.2.2 Cross-processor IPC

In the next experiment, I evaluated the performance of cross-processor IPC. The experiment used the same microbenchmark with two threads that repeatedly sent a message to each other. However, this time both threads were located on different physical processors, however on the same NUMA node. Neither processors ran any other workload aside and therefore did not have to preempt other threads or switch between address spaces.

The microbenchmark evaluated the round-trip latency of cross-processor IPC of lock-based versus message-based synchronization. The lock-based synchronization acquired a spin lock on the remote processor, transfered the message, and released the lock. Then, the receiver thread was enqueued into the remote requeue list and an IPI was sent in order to trigger a rescheduling cycle on the remote processor. The message-based synchronization required two in-kernel messages: one for initiating the IPC, and a second that the message-delivery could be started. An additional IPI may have been required to trigger the remote rescheduling cycle (see also Figure 5.3).

As expected, the latency for a message-based scheme was significantly higher

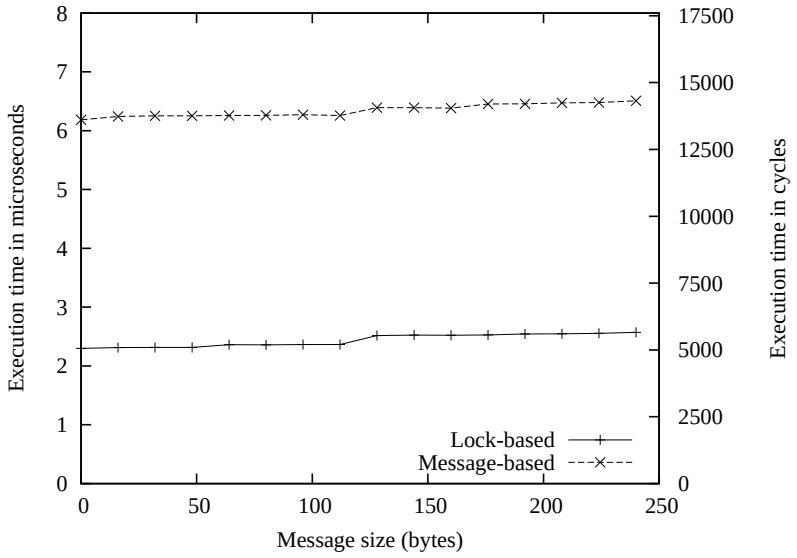


Figure 6.3: Cross-processor IPC performance between two independent processors on the same memory bus. The lock-based version used a spin-lock on the remote thread and signaled a rescheduling cycle. The lock-free version required two in-kernel RPCs per message.

than for a lock-based scheme. The latency increased between 153 to 170 percent depending on the message length. Figure 6.3 shows a detailed graph comparing the latency for both primitives.

6.2.3 Parallel Programming

An important workload for large multiprocessor systems is parallel programs. A parallel program splits one task into a set of subtasks. The subtasks can be solved on different processors in parallel and a finalization step combines the partial results. Common parallel programming environments are MPI [39] and OpenMP [10]. MPI distributes the workload via messages to workers which then compute autonomously and do not require shared memory. In contrast OpenMP fundamentally depends on shared memory between workers and uses compiler support for automated parallelizations (e.g., for loops).

A performance requirement for all parallel programming models is a low-latency and low-overhead communication mechanism across processor boundaries. I put the primary focus on the OpenMP fork-join programming model, since MPI is unlikely to base message passing on the microkernel's IPC primitive.[2] OpenMP

[2]MPI is commonly used for large scientific computations that are long running and often distributed over multiple physical machines. The workloads are explicitly parallelized and hand-optimized with the goal to maximize parallelism and minimize overall computation time. Therefore, MPI applications are unlikely to run in shared-workload environments but have complete control

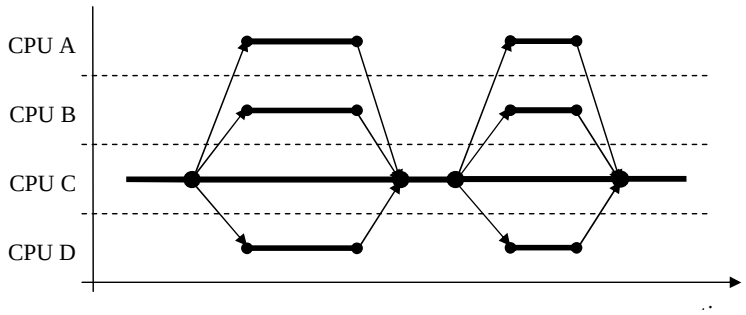


Figure 6.4: OpenMP fork-join programming model for parallel programs. CPU C executes the master thread that distributes the load and waits for completion at join points.

uses one master thread that controls the general flow of the program and a set of worker threads that execute the parallelizable subtasks. The execution model can be summarized as frequent fork–join sequences initiated by the master thread (as illustrated in Figure 6.4).

Literature suggest a variety of schemes for synchronization and workload distribution for the fork-join model. In time-sharing environments, it is generally unfeasible to have idle threads spin until new work is available, since doing so may significantly degrade the overall system performance. The significant overhead for on-demand creation of worker threads favors a model with a pool of pre-allocated worker threads. The threads are started but block if there are no outstanding requests. When new requests become available, the master unblocks the workers. Obviously, the latency of the wakeup operation is extremely performance critical.

I evaluated the latency of the IPC in the context of an OpenMP microbenchmark as proposed by Bull [23]: the PARALLEL directive. This particular benchmark evaluates the overhead of the synchronization primitives of the underlying OpenMP framework including the operating system's scheduling and signaling mechanisms. The actual executed operation is a simple mathematical computation which is irrelevant for the benchmark (but is there to avoid dead-code elimination by the compiler).

Support of a fully-fledged OpenMP suite was beyond the scope of this work. Therefore, I evaluated the performance-critical operations by a manual implementation of the test cases. The benchmark created one worker thread per processor. The worker threads entered a blocked state and waited for IPC delivery from the master thread. In order to distribute workload among the workers, the master thread sent an IPC to each of the worker threads. Afterward, the master thread waited for a notification message. The last completing thread sent a notification back to the

over the system and the scheduling regime. Simple spin-wait on a shared memory location at user level has a lower latency and is significantly more efficient than using a blocking IPC primitive. The compute time wasted for spinning is irrelevant since there are no other runnable jobs in the system.

master. The thread was determined via a shared memory variable that is decremented atomically. The following is a code sketch of the benchmark in C:

```
1  int CompletionCount;
2
3  void MasterThread()
4  {
5    CompletionCount = NumWorkers + 1;
6    for (idx=1; idx<NumWorkers; idx++) {
7      SendMessage (WorkerThreadIdOfCPU(idx));
8    }
9    DoWork();
10   if (AtomicDecrement (CompletionCount) > 0) {
11     WaitForMessage (AnyThread);
12   }
13 }
14
15 void WorkerThread()
16 {
17   while(true) {
18     WaitForMessage (MasterThreadId);
19     DoWork();
20     if (AtomicDecrement (CompletionCount) == 0) {
21       SendMessage (MasterThreadId);
22     }
23   }
```

I measured the performance of the lock-based and message-based IPC primitive for an increasing number of processors. Figure 6.5 compares the latency for both IPC types. As expected, the lock-based IPC primitive had a significant performance advantage over the message-based model. The latency increased by a factor of 2.5 for two processors up to 3.6 for sixteen processors. Note that the knee in the curve was induced by the higher memory and signaling latency when communicating over NUMA node boundaries.

The high overhead for message-based IPC can be explained by two factors. First, message-based IPC required multiple in-kernel messages and thus had a longer overall latency. Second, the lock-based IPC scheme was able to operate in parallel to the re-scheduling IPI. The master thread enqueued the worker into the requeue list and signaled the IPI. During the IPI delivery it could already start sending a message to the next worker. In the message-based IPC, the master thread was blocked for the duration of two in-kernel message deliveries (from the master's to the worker's CPU and back).

The benchmark showed the significant benefit of a lock-based IPC for latency-sensitive cross-processor signaling.

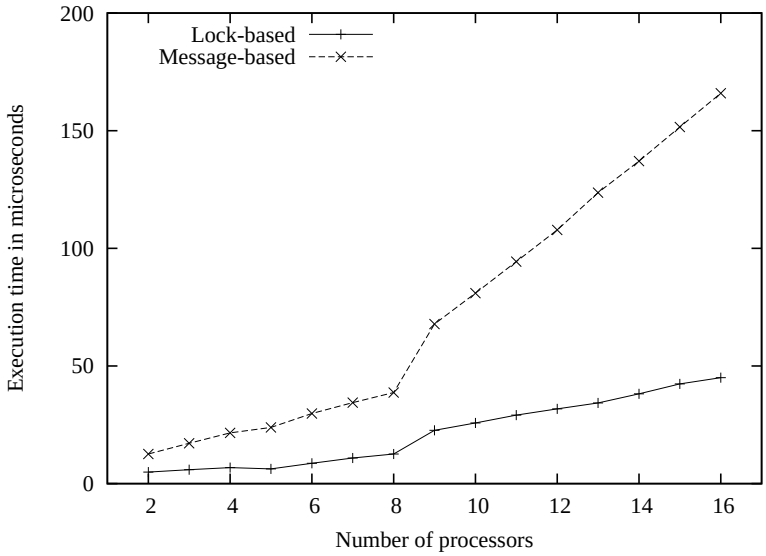


Figure 6.5: Signaling latency for fork-join programming model as used in OpenMP. The signaling latency is directly related to the overall execution time and therefore limits the speedup that can be achieved via a parallelized workload.

6.2.4 Scalability and Independence

In this benchmark, I evaluated the scalability of the IPC primitive following Unrau's design principles: (1) *preserving parallelism* in the kernel and (2) *bounded overhead* of operations independent of the number of processors. The qualitative analysis follows my extended construction principle: *preserving isolation*.

The benchmark was identical to the previously described IPC benchmark: two threads on the same processor that repeatedly sent a message to each other. This time I ran the benchmark on *multiple processors* in parallel in order to evaluate potential interference, serialization, and thus limits of scalability. The benchmark created a pair of two threads on each processor. The processors all started the benchmark synchronized. After completion of the benchmark, each thread pair reported the execution time into a processor-local log. The benchmark code was located in CPU-local memory and thus avoided interference at application level.

Figure 6.6 shows the IPC overhead for different message sizes. In addition, I varied the number of processors that run the benchmark in parallel. I started with one processor and increased up to eight. The reported execution time is the average execution time for *all active* processors. There was *no interference* and the operation scaled optimally according to Unrau's scalability requirements. The benchmark was limited to one processor context (hyperthread) per physical processor in order to avoid interference in the instruction units in SMT processors.

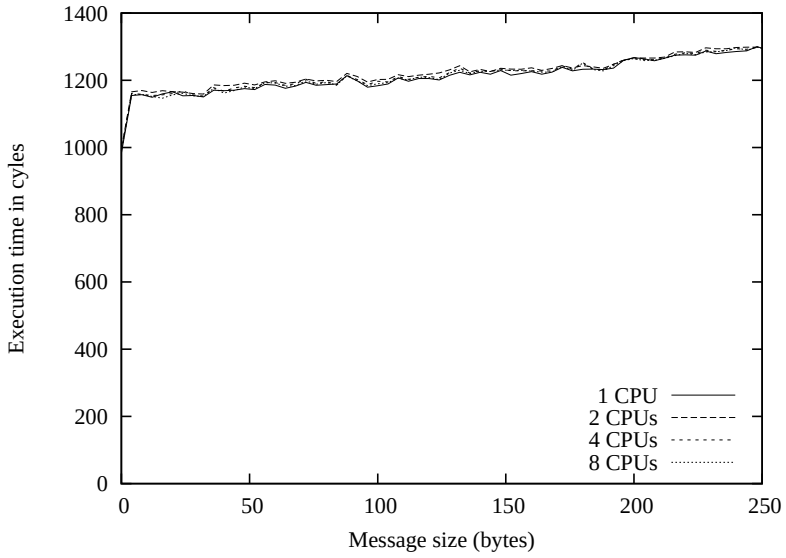


Figure 6.6: Parallel inter-address space IPC with increasing number of processors. The overhead per IPC was averaged over all processors. The curve shows no interference between the processors.

6.3 Event Logging

I evaluated the overhead and scalability of the event logging facility as described in Section 5.5.1. The benchmark addressed two aspects, the overhead of the event logging instrumentation on the kernel performance and the scalability of the operation (following Unrau's scalability requirements).

I measured the overhead of event logging on the critical IPC path. The benchmark is similar to the previous with two threads that sent a message to each other. Each thread in the system was associated with a *log domain*. The domain ID was stored in the thread control block. IPC events were only logged for communicating threads of different domains. The baseline performance was a kernel with the event logging instrumentation disabled (i.e., no logging code in the kernel).

The test for equal domains requires three additional assembler instructions on the critical code path. The actual logging code is moved off of the critical code path under the assumption that in the common case IPC will take place between threads within the same domain. Each domain has an associated counter and a reference to a log control field. The log control field specifies which values are logged and contains a reference to the log buffer. The log buffer size was 128 Bytes.

Figure 6.7 and Figure 6.8 show the overhead for different logging configurations on the 8way Xeon 2.2 GHz. The additional domain test for the common case incurred a negligible overhead and for the inter-address space even outperformed the non-instrumented code path. In order to achieve comparable results I replaced the logging test with two 6-byte no-op instruction; otherwise, the kernel binaries

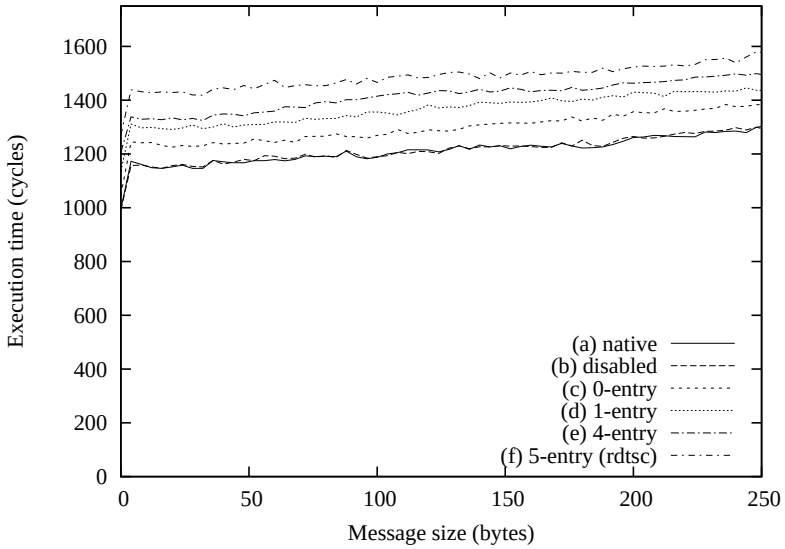


Figure 6.7: Logging overhead for Intel Xeon 2.2GHz for inter-address space IPC. (a) base-line performance, kernel contains no logging code, (b) common case, kernel contains logging code, but threads belong to the same domain and no log is recorded, (c) execute accounting code but without writing data into the log file, (d) log of the one entry (4 bytes), (e) log of four entries (16 bytes), (f) log of five entries (20 bytes) including the processor's time-stamp counter using rdtsc.

were identical.

I further evaluated the scalability of the logging facility. Similar to the IPC scalability benchmark in Section 6.2.4, I ran multiple parallel threads to evaluate the independence and potential interference. The reported values are the average cost per IPC for messages with increasing size starting at 0 bytes to 252 bytes. As expected, the logging code did not influence scalability and the overhead remained constant for all processors and was identical to the overhead for a single processor (see Figure 6.9).

6.4 Memory Management

I evaluated the overhead of the unmap primitive using coarse-grain vs. fine-grain locks as described in Section 5.4. I created a test task with one thread in an address space that mapped 1024 4KByte pages out of one 4MByte page. In the first case, each mapping was protected by an individual lock while in the second case all mappings were protected by one coarse lock. In the benchmark I evaluated the latency for repeatedly removing the write permissions from the page using *unmap*. The unmap operation walks the page table and performs a lookup of the associated map node, acquires the lock and then clears the permissions.

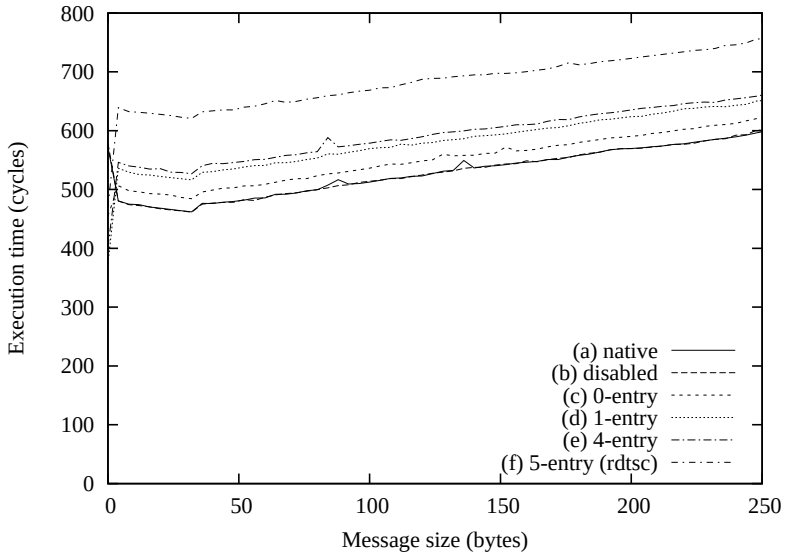


Figure 6.8: Logging overhead for Intel Xeon 2.2GHz for intra-address space IPC. (legend see Figure 6.7.) The benchmarks for (a) native and (b) disabled logs show an anomaly for the zero-byte message that I attributed to trace-cache rebuilding on the Pentium 4 [29].

I choose this operation as a general representative for unmap because it allowed the benchmark to repeatedly perform the same operation on the mapping database thereby avoided side effects such as cache effects or different map node allocations. When revoking write permissions, the kernel performs exactly the same operations as a normal unmap except for finally freeing the map node. Freeing the map node only adds a static runtime overhead; using local memory pools makes the operation independent of other processors.

6.4.1 Locking Overhead

The benchmark repeatedly revoked write permissions via the unmap operation with a varying number of pages (power of two). The benchmark repeated the operation 200 times and calculated the average cycles per mapping and the 95% confidence interval. The overhead for entering and exiting was the dominating cost of the operation when modifying only a few pages.[3] When the benchmark unmapped a larger number of pages (e.g., required for implementing copy-on-write for *fork*

[3]IA-32 supports two different methods for entering and exiting the kernel: sysenter and int. The sysenter operation is highly optimized and almost an order of magnitude faster than int. In the current implementation of L4Ka::Pistachio sysenter is only used for the IPC primitive. All other system calls — including unmap — use the slower int operation. The main reason for this design decision was to maximize the available registers for the IPC operation. An alternative and already considered system-call encoding would drastically reduce the base overhead for unmap.

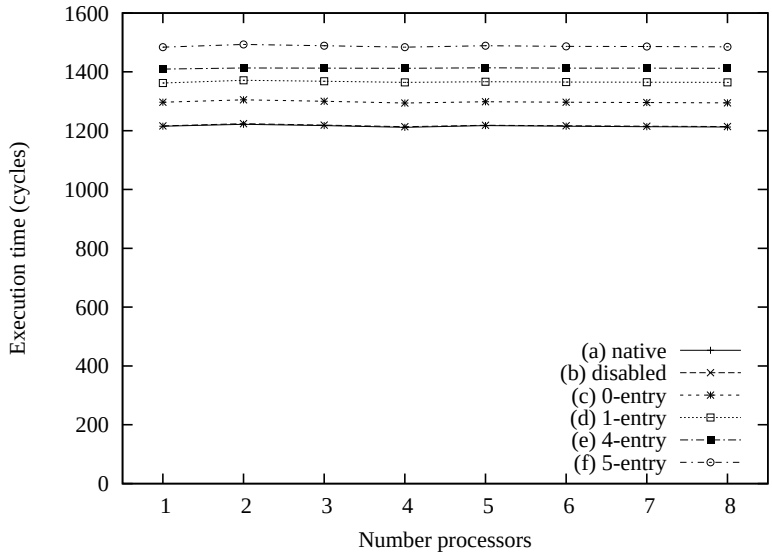


Figure 6.9: Average execution time of parallel inter-address-space IPC with event logging for increased number of processors. The reported values are the average costs for messages of 0 bytes to 252 bytes.

Application	Apache 2	MySQL	sshd	Emacs	bash
Pages	2661	4453	1043	1513	185

Table 6.2: Page working sets for a set of common Linux applications. The values are derived from a standard Suse 9.1 Linux system.

or deletion of an address space) the lock overhead accounted for 185 cycles per mapping, which is a 41% runtime overhead. Figure 6.10 depicts the measured overhead for coarse-grain and fine-grain locking for the different number of pages.

The benchmark showed, that for moderately populated address spaces with only 64 pages (identical to 256KByte memory), the runtime overhead was 15 percent, 24 percent for 128 pages and 32 percent for 256 pages. For comparison, Table 6.2 lists the number of mappings for some common Linux applications. Even the smallest application — the shell — has a memory working set of 185 pages.

Additionally, the individual locks incur a higher cache footprint with one cache line per lock (in order to avoid false sharing) increasing the cache footprint of the data cache.

6.4.2 Scalability

In another benchmark, I evaluated the scalability of the two locking schemes. Parallel to the previously described task, I created additional tasks on each processor that maps one of the memory pages into its address space. In the first case, the

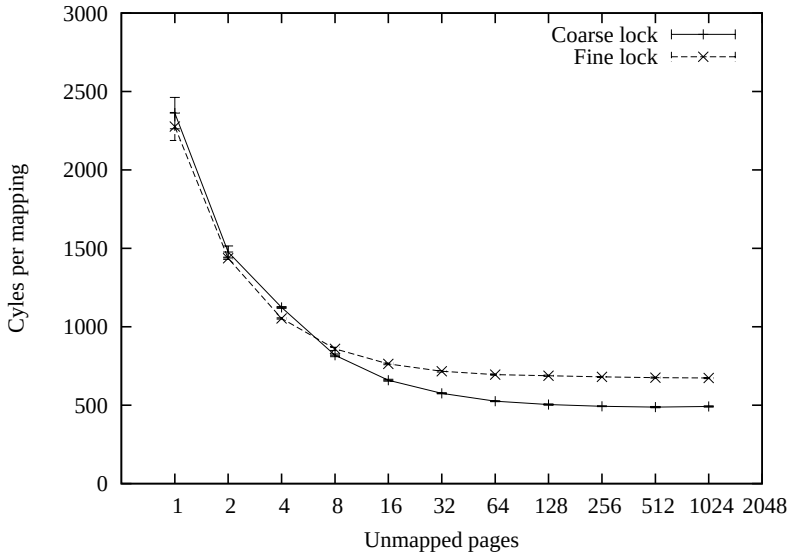
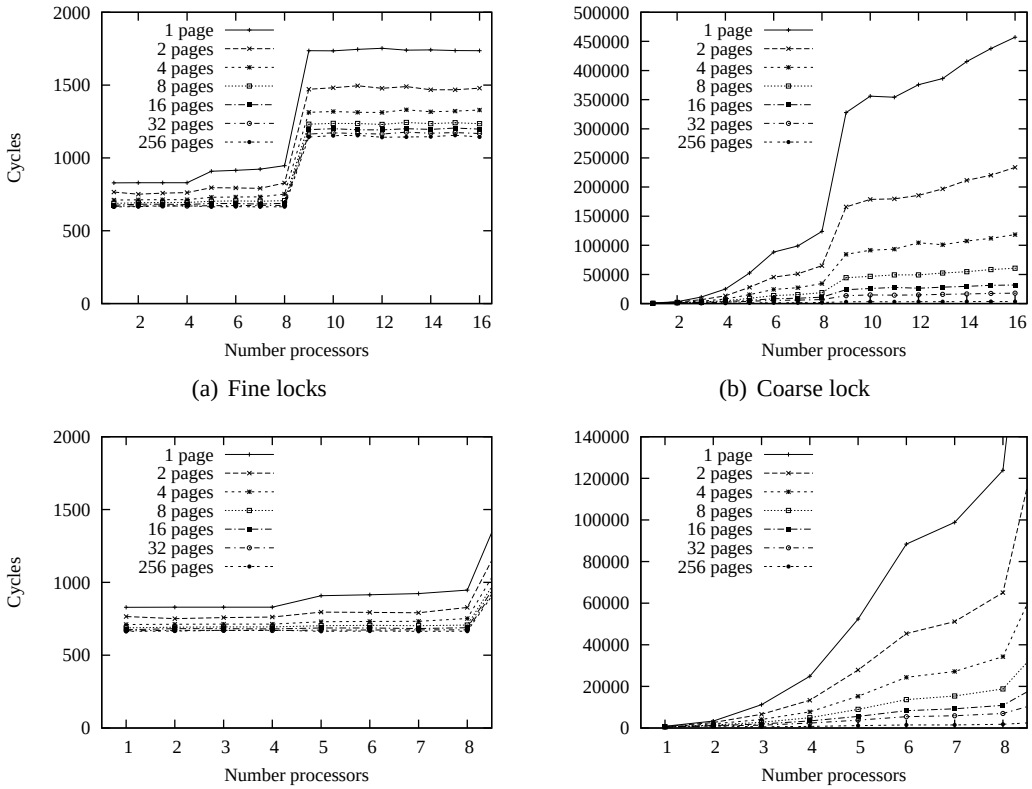


Figure 6.10: Runtime overhead for unmap per page for varying number of pages using coarse-grain and fine-grain locks.

memory was mapped with an individual lock per mapping. In the second case, all page mappings were protected by one coarse lock. I evaluated the scalability of the mechanism when manipulating permissions to the same memory page in parallel on multiple processors.

In the benchmark I derived two sets of performance results. First, I evaluated the runtime overhead per mapping for multiple mappings. Different to the previous benchmark, I batched 32 unmaps in a single system call reducing the runtime overhead for entering and exiting the kernel and also the runtime deviation. I compared the costs per mapping for an increasing number of parallel operating unmaps with coarse-grain and fine-grain locks, shown in Figure 6.11.

With fine-grain locks, the cost per mapping remained almost constant, independent of the number of processors (see Figure 6.11a). The graph shows two anomalies, one between 4 and 5 processors and a second between 8 and 9 processors. In the benchmark I placed the parallel unmap operations first on physical processors (1 to 8) followed by logical processors (9 to 16). The first knee at 5 processors is when the unmap operation hit the first NUMA processor (5 to 8). The overhead reduced with an increased number of unmapped pages, since the cache-line transfer was only necessary for one page out of the overall set of pages. The second, more drastic overhead increase was incurred by parallel execution of multiple processor threads of the same physical processor (SMT). The two processor threads competed for execution resources and interfered with each other. While the overhead per mapping increased by a factor of two, the cost per mapping remained constant for 9 to 16 parallel operating unmaps. Figure 6.11(c) shows an enlarged graph for the first 8 (physical) processors.



(a) Fine locks

(b) Coarse lock

(c) Fine locks (8 processors)

(d) Coarse lock (8 processors)

Figure 6.11: Unmap performance for one processor repeatedly unmapping a set of pages. (a) and (c) show the overhead with fine granular mapping while (b) and (d) show the same benchmark with coarse granular locking.

The same benchmark with coarse-grain locking showed very different results. The per-mapping overhead increased steeply with the number of parallel operating processors. As described, the microkernel's unmap algorithm performs critical section fusing if two consecutive mappings are protected by the same lock. Thus, with an increasing number of pages, the overall number of acquired locks decreases and thus the *waiting time* for the lock. Figure 6.11(b) depicts the cost per mapping for an increasing number of processors. Compared to the single processor case, the overhead for a single page unmap increased by a factor of 3.8 for two processors, 13.2 for three processors, and up to a factor of 537.3 for sixteen parallel processors. For 64 pages, the overhead increased by a factor of 1.93 for two processors, 1.96 for three processors, up to a factor of 20.4 for sixteen processors.

In the same benchmark, I also measured the overhead for each individual processor unmapping a page. Each processor ran a task that repeatedly tried to unmap one page from its address space that is shared with the primary task. In the first case, the page mapping was protected by a separate lock and in the second case all

page mappings were protected by the same lock.

Figure 6.12 compares the overheads for the single-page unmap for an increasing number of parallel operations with coarse-grain and fine-grain locking. The *x* axes denote the number of processors. The figures show individual graphs for different number of unmapped pages *of the primary task*. Since the primary task performed critical section fusing, the average lock holding time increased. For coarse locking, the longer lock holding time reflected in a longer lock waiting time for the parallely operating single page unmaps. The overall cost for the operation was determined by having all processors time the latency of 200 unmap operations. Afterward, the total execution time for all processors was summed up and divided by the total number of unmap invocations.

Figure 6.12(a) shows that the cost for unmaps remained almost constant for up to eight processors. With nine processors, the cost per mapping increased which has to be attributed to interference of processor threads in the same processor. Figure 6.12(c) shows the enlarged version of the graph for the first eight processors.

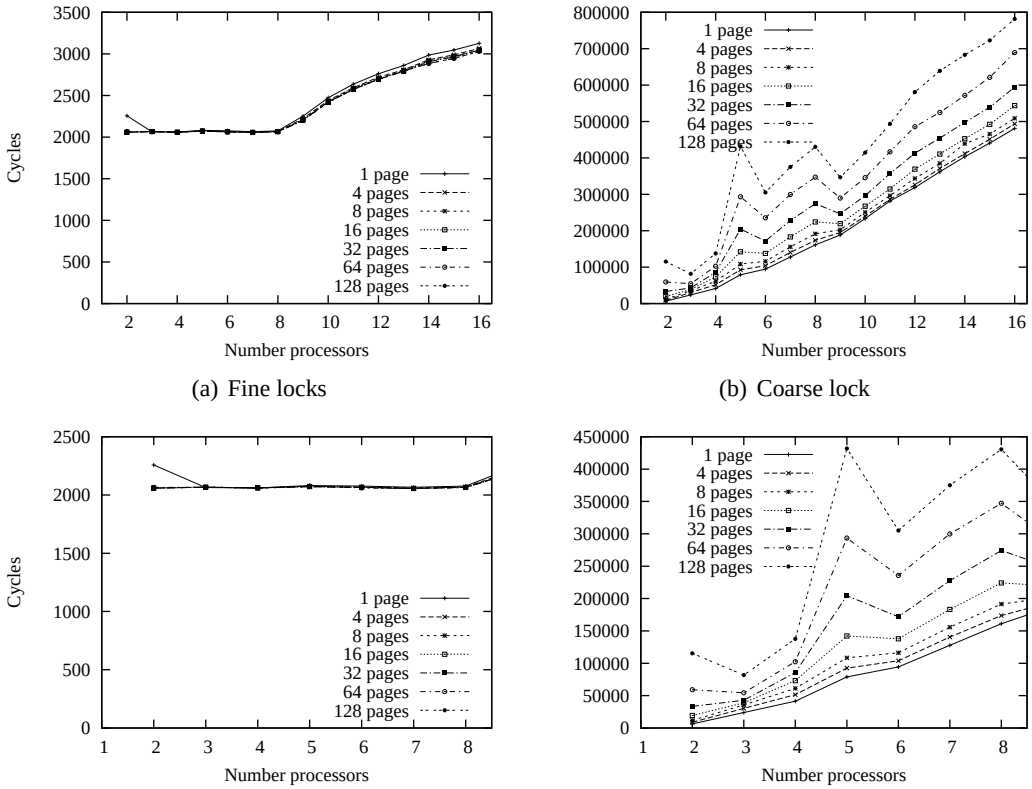
In the case of coarse locking, the overhead for unmapping increased significantly for five processors. A more detailed analysis of the overhead for individual processors revealed that the reason for the drastic increase was unfairness in the memory subsystem of the NUMA configuration. At first, it appeared that the fifth processor starved on the lock and the other four processors (that were located on the same NUMA node) had a higher probability to acquire the lock. However, this would not explain the significant decrease of the six processor configuration. Actually, in the configuration with five processors, the average lock acquisition latency was almost identical for all processors but twice as high as in the configuration with four processors (40K cycles vs. 78K cycles).

With six processors the situation changed; the configuration had four active processors on one NUMA node and two on the second node. The two processors on the second NUMA node had a significantly higher probability for lock acquisitions due to the lower lock competition. After the lock was released by CPU 5 there was a high chance CPU 6 would immediately acquire it. While the average unmap time per mapping increased to 123K cycles for CPU 2 to 4, it decreased to 52K for CPU 5 and 6. The lock effectively ping-ponged between processor 5 and 6. With the significantly lower cost for those two processors, the overall cost per mapping decreased compared to the five CPU configuration.

6.5 Summary

In this section, I evaluated the performance of dynamic lock adaptation and tracking of parallelism applied to the L4Ka::Pistachio microkernel. Dynamic locks are used on the critical IPC path and for the virtual memory subsystem. The benchmarks confirmed the initial hypothesis that a single synchronization mechanism is insufficient for the microkernel and an adaptive scheme is required instead.

Summarizing, the following performance has been observed:



(a) Fine locks

(b) Coarse lock

(c) Fine locks (8 processors)

(d) Coarse lock (8 processors)

Figure 6.12: Average unmap cost for parallel unmaps of independent pages on different processors. Fine granular locks [(a) and (c)] show linear speedup, while for coarse locking the overhead increases drastically [(b) and (d)].

- The runtime overhead of locks for inter-address space IPC accounted for 16 to 20 percent compared to a lock-free variant. For intra-address space IPC the lock overhead was between 35 to 64 percent. The overhead was eliminated by dynamic lock elimination.

- The overhead of message-based synchronization for cross-processor IPC compared to the lock-based solution was 153 to 170 percent depending on the message length.

- The cost of IPC on the same processor remained constant with an increasing number of processors in the system and thus scales. All accessed kernel data structures are local to the processor.

- The event logging scheme provided kernel resource information to applications. The additional runtime overhead when logging was disabled (i.e., a test for a configuration register) induced negligible overhead. In some cases,

the Pentium 4 microarchitecture even provided better performance than without the additional logging instrumentation. The overhead for the operation was constant and independent of other processors and thus scales.

- Benchmarks of the virtual memory subsystem showed a runtime overhead for fine-grain locks between 15 and 37 percent as compared to coarse-grained locks (not accounting the higher cache footprint for locks). Dynamic adaptation of the lock granularity was able to eliminate that overhead.

- When multiple threads concurrently revoked page permissions, fine granular locks combined with the TLB invalidation mechanism scaled linearly up to the number of available *physical* processors. For SMT threads the overhead increased due to instruction interference but with a constant overhead. With coarse locks, the average costs per page mapping increased between a factor of two up to three orders of magnitude.

In the following chapter, I summarize my work and address future work.

Chapter 7

Conclusion

This chapter concludes the dissertation with a summary of its contributions followed by suggestions for future work. Finally, I give some concluding remarks.

7.1 Contributions of This Work

In this dissertation, I addressed multiprocessor scalability of microkernel-based systems. I developed methodologies to strictly separate scalability and performance-relevant synchronization and coherency schemes from the resource management policies. Such strict separation is a novel approach and has not been considered in previous work.

My solutions comprise four main contributions:

Tracking of parallelism. (Section 4.2)

I developed a tracking scheme for resource usage: the *processor cluster mask*. The cluster mask uses a space-efficient encoding that is independent of the number of processors in the system. It is used for permission and TLB dirty tracking and is the fundamental building block for the remaining contributions.

Adaptive lock primitive. (Section 4.3)

I developed a *dynamic lock* primitive that—depending on the degree of parallelism—can be dynamically and safely enabled and disabled at runtime. I apply the fundamental principle of read-copy update epochs to a new domain: dynamic instruction adaptation.

By cascading multiple dynamic locks, applications can safely adjust the *lock granularity* of kernel objects. Hereby, dynamic locks enable *critical section fusing* and *critical section splitting* at runtime.

TLB Coherency Epoch. (Section 4.4)

I developed a TLB coherency tracking scheme that decouples page permission updates from the outstanding TLB shoot-downs of remote processors. Parallel page-permission updates can be completed by initiating the TLB update from remote. My approach allows the combination of expensive TLB coherency updates in a single remote shoot-down while still providing fine granular synchronization on memory objects. In Section 4.4.2, I propose a specific variant for IA-32, the *TLB version vector*.

Event Logging. (Section 4.5)

In order to efficiently transport resource-usage information between the kernel and a user-level scheduler and also between isolated operating system components, I developed a configurable event logging scheme. By using per-processor logs and providing fine-granular control over the logged data, it is possible to provide detailed resoure usage information to schedulers in an efficient and scalable manner.

I validated my proposed design with the L4Ka::Pistachio microkernel. The primary directions and results of this work have been influenced by research I undertook, that has not been specifically addressed in this dissertation. Because of this, I want to briefly mention it here.

In order to validate the general design of a multiprocessor microkernel, I developed a multiprocessor para-virtualization environment based on the Linux kernel. Multiple Linux instances serve as a complex application workload on L4Ka::Pistachio that stresses the kernel's primitives and is extremely performance sensitive. In the context of VMs, I developed an efficient method to avoid excessive lock spinning times due to preemption of virtual processors of a multiprocessor environment. Furthermore, I developed a scheme to load-balance the per-VM workload considering the overall load situation. The load-balancing scheme compensates for differing processor allocations for one VM and is an important application of the event-logging scheme described in Section 4.5. The results of this work are detailed in [111].

7.2 Suggestions for Future Work

I see four major areas for future work:

First, I primarily focused on one architecture with specific performance properties: IA-32. The trade-offs for other hardware architectures are substantially different and need a further investigation. In particular, IA-32's strong memory ordering model results in a very high synchronization penalty which may be less substantial on other processor architectures.

Second, current developments in the area of SMT and multicore systems are changing the hardware properties. In particular tight integration of caches and improved IPI delivery latency reduce overheads for inter-processor interaction. These

new architectural properties also need to be reflected in the microkernel interface. For example, the strict separation of processor threads in SMT systems could turn out to be overly restrictive and a granularity that addresses processor cores may be more appropriate.

Third, the fundamental idea of alternative lock primitives that depends on the degree of parallelism is currently implemented as a software solution. The software-based approach requires additional code and thus incurs runtime overhead. The overhead could be eliminated by having explicit architectural support by processors. The architectural support could include optimized operations on the cluster mask (such as simple tests and merge) and support for dynamic locks. For example, a lock primitive or memory operation could carry a coherency identifier that, depending on the processor isolation mask, serves as a filter for coherency traffic in the memory (or cache) subsystem.

Fourth, the new microkernel primitives require extensive testing and evaluation in a variety of real-world scenarios including large-scale databases. While I showed low overhead and independence for individual kernel primitive it remains open how the kernel behaves in more complex environments. The camouflage of microkernels as *virtual machine monitors* or *hypervisors* bring many prevalent microkernel issues to industry today. The construction methodologies differ insofar, that fine-granular decomposition of a monolithic system is of less importance. Nevertheless, the general mechanisms for controlling scalability from application level is as relevant and applicable to a wide range of resource management problems.

7.3 Concluding Remarks

In this thesis I have described a methodology for adjusting scalability-relevant parameters of multiprocessor microkernels. The shift towards highly parallel processor architectures is at its beginning and one can expect more drastic increases of processor contexts. This change includes all areas of computing starting from embedded devices to large-scale servers. Furthermore, the increasing complexity of operating systems and different business demands requires for radically differing OS structures. Virtual machines and highly customized appliance-like systems will be common place.

Such systems require a flexible and efficient microkernel that shows excellent performance and scalability. I developed a set of mechanisms that lay the ground for construction of scalable systems on top of a microkernel. I validated and evaluated it with L4Ka::Pistachio, a widely used and versatile microkernel developed at the University of Karlsruhe.

Bibliography

[1] The DragonFly BSD Project. `http://www.dragonflybsd.org`, (accessed March 2005).

[2] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[3] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings division of Addison Wesley Inc., Redwood City, CA, 2nd edition, 1994.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Proceedings of the SJCC*, 31:483–485, 1967.

[5] E. Anderson, B. N. Bershard, E. D. Lazowska, and H. M. Levy. Scheduler activation: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1), 1993.

[6] T. E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume II - Software, University Park, Penn, August 1989.

[7] Jonathan Appavoo. *Clustered Objects*. PhD thesis, University of Toronto, 2005.

[8] Jonathan Appavoo, Marc Auslander, Dima DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, et al. K42 overview. Whitepaper, August 2002.

[9] Jonathan Appavoo, Marc Auslander, Dima DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, et al. Providing a Linux API on the scalable K42 kernel. In *Freenix*, 2003.

[10] The OpenMP ARB. OpenMP. `http://www.openmp.org`.

[11] J. Archibald and J-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[12] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill framework for virtual memory diversity. In *8th Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001.

[13] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[14] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software Practice & Experience*, 15(9):901, September 1985.

[15] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, October 2003.

[16] James M. Barton and Nawaf Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 45–69. Springer-Verlag, 1995.

[17] F. Bellosa and M. Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):1–2, August 1996.

[18] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain Resort, CO, December 1995.

[19] Brian N. Bershad, Thomas E. Anderson, Lazowska Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *12th ACM Symposium on Operating Systems Principles (SOSP)*, volume 23, pages 102–113, October 3–6 1989.

[20] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer Magazin*, 23(5):35, May 1990.

[21] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Santa Fe, New Mexico, 20–22 November 1994. IEEE.

[22] Bryan R. Buck and Jeffrey K. Hollingsworth. Using hardware performance monitors to isolate memory bottlenecks. In *Proceedings of Supercomputing'2000*, Dallas, TX, November 2000. UMCP.

[23] J. Mark Bull and Darragh O'Neill. A microbenchmark suite for openMP 2.0. In *3rd European Workshop on OpenMP*, September 2001.

[24] Henry Burkhardt, III, S. Frank, B. Knobe, and James Rothnie. Overview of the KSR1 computer system introduction to the KSR1. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.

[25] Richard P. Case and Andris Padegs. Architecture of the IBM System/370. *Communications of the ACM*, 21(1):73–96, January 1978.

[26] Eliseu M. Chaves, Jr., Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. In *The Symposium on Experiences with Distributed and Multiprocessor Systems*, Atlanta, GA, March 1991.

[27] Benjie Chen. Multiprocessing with the Exokernel operating system. Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2000.

[28] BBN Advanced Computers. *Inside the Butterfly-Plus*. Cambridge, MA, October 1987.

[29] Intel Corp. *IA-32 Intel Architecture Software Developer's Manual, Volume 1–3*, 2004.

[30] Intel Corp. *Intel XScale Core, Developer's Manual*, 2004.

[31] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[32] D. H. Brown Associates, Inc. Unisys ES7000 challenges Sun E10000. Whitepaper, `http://www.microsoft-sap.com/docs/brown_white_paper.pdf`, September 2000.

[33] L. Van Doorn, P. Homburg, and A. S. Tanenbaum. Paramecium: An extensible object-based kernel. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 86–89, Orcas Island, WA, May 1995.

[34] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating Systems Principles*

(SOSP), pages 251–266, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[35] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems. IBM Research Report RC 19790 (87657), August 1997.

[36] Dror G. Feitelson and Larry Rudolph. Coscheduling Based on Run-Time Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):136–160, April 1995.

[37] Dror G. Feitelson, Larry Rudolph, Schwiegelshohn Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[38] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, California, 1–3 May 1978.

[39] MPI Forum. MPI: A message-passing interface standard. *The international Journal of Supercomputing and High Performance Computing*, pages 159–416, feb 1994.

[40] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999.

[41] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[42] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill Multiserver Approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[43] Robert P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[44] Ruth E. Goldenberg and Saro Saravanan. *VMS for Alpha Platforms—Internals and Data Structures*, volume 1. DEC Press, Burlington, MA, preliminary edition, 1992.

[45] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 24, Boston, MA, April 1989.

[46] Allan Gottlieb and Clyde P. Kruskal. Coordinating parallel processors: A partial unification. *ACM Computer Architecture News*, October 1981.

[47] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, 1999.

[48] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 123–136, Berkeley, October 28–31 1996. USENIX Association.

[49] System Architecture Group. *L4 X.2 Reference Manual*. University of Karlsruhe, Germany, 6 edition, June 2004.

[50] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. Technical Report CSL-TR-90-417, Computer Systems Lab, Stanford, University, Stanford, CA, March 1990.

[51] Andreas Haeberlen. Managing kernel memory resources from user level. Master's thesis, University of Karlsruhe, `http://i30www.ira.uka.de/teaching/theses/pastthesis`, 2003.

[52] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, September 24–26 2003.

[53] Claude-Joachim Hamann, Frank Mehnert, Hermann Härtig, Jean Wolter, Lars Reuther, Martin Borriss, Michael Hohmuth, Robert Baumgartl, and Sebastian Schönberg. DROPS OS support for distributed multimedia applications. In *8th SIGOPS European Workshop*, Sintra, Portugal, December 11 2001.

[54] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of microkernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, SaintMalo, France, October 1997.

[55] C.-T. Ho. *Optimal communication primitive and graph embeddings on hypercubes*. PhD thesis, Yale University, 1990.

[56] C.-T. Ho and L. Johnsson. Distributed routing algorithm for broadcasting and personalized communication in hypercubes. In *Proc. 1986 Int. Conf. Par. Proc.*, pages 640–648, 1986.

[57] Michael Hohmuth. The Fiasco kernel: System architecture. Technical Report TUD-FI02-06-Juli-2002, TU Dresden, 2002.

[58] Michael Hohmuth. *Pragmatic Nonblocking Synchronization for Real-Time Systems*. PhD thesis, Technische Universität Dresden, October 2002.

[59] IEEE. *IEEE Std 1596–1992: IEEE Standard for Scalable Coherent Interface*. IEEE, Inc., August 1993.

[60] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 29–30 1999.

[61] E. H. Jensen, G. W Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report 212–157, Lawrence Livermore Laboratory, 1987.

[62] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 5–8 1997.

[63] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 170–180, New York, June 2–4 1997. ACM Press.

[64] A. Langerman, J. Boykin, S. LoVerso, and S. Mangalat. A highly-parallelized Mach-based Vnode filesystem. *USENIX*, pages 297–312, Winter 1990.

[65] Edward D. Lazowska. *Quantitative System Performance, Computer System Analysis Using Queuing Network Models*. Prentice-Hall Inc, Englewood Cliffs, NJ 07632, 1984.

[66] F. Lee. Study of 'look aside' memory. *IEEE Transactions on Computers*, C-18(11), November 1969.

[67] C. E. Leierson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, c-34:892–901, October 1985.

[68] Daniel Lenoski, James Laudon, Gharachorloo Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[69] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[70] Jochen Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, December 1993.

[71] Jochen Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, December 1995.

[72] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The L4Ka vision, April 2001.

[73] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th Workshop on Hot Topics in Operating Systems (HotOS)*. IBM, May 1997.

[74] Jochen Liedtke, Nayeem Islam, and Yoonho Park. Preventing denial of service attacks on a microkernel for WebOSes. In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, May 1997.

[75] Jochen Liedtke, Vsevolod Panteleenko, Trent Jaeger, and Nayeem Islam. High-performance caching with the Lava Hit-Server. In *USENIX 1998 Annual Technical Conference*, June 1998.

[76] Beng-Hong Lim. *Reactive synchronization algorithms for multiprocessors*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1995.

[77] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report ISRN SICS-R–94-07–SE, Swedish Institute of Computer Science, 1994.

[78] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS–005–91, Columbia University, 1991.

[79] Paul E. McKenney. Selecting locking primitives for parallel programming. *Communications of the ACM*, 39(10):75–82, 1996.

[80] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read-copy update. In *The Ottawa Linux Symposium*, June 2002.

[81] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998.

[82] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[83] Microsoft. Authenticode technology. Microsoft's Developer Network Library, October 1996.

[84] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 39–51, 1987.

[85] George C. Necula. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, Paris, January 1997.

[86] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *2nd Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 1996.

[87] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56, March 1991.

[88] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[89] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, St. Charles, IL, August 1985.

[90] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, Zurich Laboratory, April 2001.

[91] Daniel Potts, Simon Winwood, and Gernot Heiser. Design and implementation of the L4 microkernel for Alpha multiprocessors. Technical Report UNSW-CSE-TR-0201, University of New South Wales, Australia, February 10 2002.

[92] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 294–305, Austin, Texas, December 1–5, 2001. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[93] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[94] Stuart Ritchie. The Raven kernel: a microkernel for shared memory multiprocessors. Technical Report TR-93-36, University of British Columbia, Department of Computer Science, 30 April 1993.

[95] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, Ann Arbor, Michigan, June 5–7, 1984. IEEE Computer Society and ACM SIGARCH.

[96] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, 1991.

[97] Sebastian Schönberg. *Using PCI-Bus Systems in Real-Time Environments*. PhD thesis, University of Technology, Dresden, 2002.

[98] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, Kiawah Island, SC, 1999.

[99] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 106–126. Springer-Verlag, 1995.

[100] Jan Stoess. Using operating system instrumentation and event logging to support user-level multiprocessor scheduler. Master's thesis, University of Karlsruhe, Germany, April 2005.

[101] Harold S. Stone. *High-Performance Computer Architecture*. Addison-Wesley, 2nd edition, 1990.

[102] Sun, Inc. Sun enterprise 10000 server. `http://www.sun.com/servers/highend/e10000`, (accessed March 2005).

[103] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.

[104] Cristan Szmajda. Calypso: A portable translation layer. In *2nd Workshop on Microkernel-based Systems*, `http://www.disy.cse.unsw.edu.au/publications.pml`, Lake Luise, Canada, October 2001.

[105] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, 1999.

[106] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[107] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 272–274, May 1993.

[108] Dean Tullsen, Susan Eggers, and Henry Levy. Simultanious multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

[109] Volkmar Uhlig. Design rationale for L4 on multiprocessors. Technical report, University of Karlsruhe, Germany, May 2005.

[110] Volkmar Uhlig, Uwe Dannowski, Espen Skoglund, Andreas Haeberlen, and Gernot Heiser. Performance of address-space multiplexing on the Pentium. Technical Report 2002-01, University of Karlsruhe, 2002.

[111] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 43–56, San Jose, CA, May 6–7 2004.

[112] Unisys, Inc. ES7000 Servers. `http://www.unisys.com/products/es7000_linux`, (accessed March 2005).

[113] Ronald Unrau. *Scalable Memory Management through Hierarchical Symmetric Multiprocessing*. Ph.D. thesis, University of Toronto, Toronto, Ontario, January 1993.

[114] Ronald Unrau, Michael Stumm, and Orran Krieger. Hierarchical clustering: A structure for scalable multiprocessor operating system design. Technical Report CSRI-268, University of Toronto, March 1992.

[115] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 139–152, Berkeley, CA, USA, November 1994. USENIX Association.

[116] Marcus Völp. Design and implementation of the recursive virtual address space model for small scale multiprocessor systems. Master's thesis, University of Karlsruhe, `http://i30www.ira.uka.de/teaching/theses/pastthesis`, 2002.

[117] T. H. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. Ph.D. thesis, Computer Science, Graduate Division, University of California, Berkeley, CA, 1993.

[118] Zvonko Vranesic, Michael Stumm, Ron White, and David Lewis. The Hector multiprocessor. *IEEE Computer Magazin*, 24(1), January 1991.

[119] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[120] Boris Weissman. Performance counters and state sharing annotations: A unified approach to thread locality. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 3–7, 1998.

[121] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *SC2003: Igniting Innovation*, Phoenix, AZ, November 2003.

[122] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[123] Michael Wayne Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. PhD thesis, Carnegie-Mellon University, 1989.