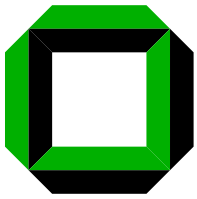

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Using Operating System Instrumentation and Event Logging to Support User-level Multiprocessor Schedulers

Jan Stöß

Diplomarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inf. Volkmar Uhlig

24. März 2005

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 24. März 2005

_____

Jan Stöß

Abstract

This work presents a novel approach to support multiprocessor scheduling at user-level. We propose to use system instrumentation and event logging to obtain run-time information relevant for scheduling, from different system sources such as the hardware, operating system components, or application programs.

We instrument the system components with event log handlers that record scheduling data in designated log files held in memory. The user-level scheduler can analyze and evaluate these log files to base its processor allocation policies on. Our approach allows for flexible, efficient, and scalable accumulation of scheduler-relevant run-time characteristics; it was designed to be applicable both to user-level components, and to the safety-critical operating system kernel. To demonstrate our approach, we have developed a prototype logging infrastructure within a real-world, microkernel-based, component operating system. We successfully extract several run-time scheduling characteristics from different components, including timing properties and communication patterns from the microkernel, and shared-memory usage from a user-level device driver subsystem. Measurements indicate an undramatic overhead for logging, with baseline logging costs between about 26 and 275 clock cycles on an Intel 2.2 GHz Xeon Processor, and application-level overhead on driver throughput of about 3.8 percent for logging kernel-based communication, and about 10.5 percent for logging shared memory usage.

Acknowledgments

First, and foremost, I want to thank my advisors, Volkmar Uhlig and Prof. Dr. Frank Bellosa. Without their assistance, encouragement, and expertise, this work would not have been possible.

I would like to thank all members of the System Architecture Group at Karlsruhe, for providing a productive, inspiring, and collegial work environment. Further I want to thank Pedram Azad and Gerd Liefländer for proofreading my thesis, and Joshua LeVasseur for patiently answering my questions about his driver framework. Thanks go also to Sebastian Biemüller, for his help in aesthetic questions.

Special gratitude is reserved for my parents, Anne and Karl-Hans Stöß, for all their selfless support throughout my life and my studies. Finally, I want to thank Katrin Schulz, for her patience, for her support, and for bringing joy to my life.

Contents

Chapter 1

Introduction

Moving resource management out of the operating system kernel has several advantages over an in-kernel solution. Application-controlled management of resources increases the flexibility of a system, by making room for domain-specific solutions and adaptations. Also, it enhances robustness and safety, since it reduces the privileged and mandatory part of the system. While user-level resource management is a promising approach for other resources, too, it is particularly interesting within the context of *multiprocessor scheduling*.

It is one of the core responsibilities of a multiprocessor operating system to allocate the available processors among the concurrent activities and programs. The decision how processor allocation should take place depends on a variety of system parameters – be it architectural properties, like the costs of inter-processor communication, be it workload characteristics, like processor service demands of applications, or be it user-specified classifications, like the desired priority of a particular activity. With the diversity of computer hardware, application workloads, and users' expectations comes along a diversity of scheduling strategies and implementations. In fact, there exists a plethora of such approaches, suitable for different environments and circumstances [23]. With a scheduler subsystem that is integrated within the kernel and therefore hard to replace, a system will inherently be limited to a subset of policies, as it will be applicable only to specific environments. Making the scheduling logic extensible, by moving it out of the kernel, can therefore greatly enhance the system's adaptiveness and tailorability to different environments. Moreover, keeping the scheduler in a deprivileged component ensures, that a faulty or malicious scheduler implementation is confined within a protection domain, and can not compromise the entire system.

However, such a modular scheduler architecture poses a new challenge on system design. Scheduling is principally based on information. While some of the deciding factors are more of static nature, like for example user-specified priorities, or static application characteristics, many of them accrue *dynamically*, that is, at run time of the system. In contrast to static information, dynamic scheduling properties must be gathered and evaluated while the system runs. A subsystem running in privileged mode principally has access to all system and hardware data. Kernel-level schedulers can therefore easily obtain the information they require to perform their policies and decisions. However, for a scheduler running at user mode, with restricted access to protected data, accumulation and extraction of scheduling information becomes a very challenging task.

From the perspective of a user-level scheduler, the required information is *dis-*

tributed among different parts of the system. Sources for scheduler-relevant information may be the processor hardware, operating system components, or the application workload. Moreover, the deprivileged scheduler does not necessarily have direct access to the required scheduling information, since this information resides in unaccessible hardware registers, or different protection domains. Furthermore, the characteristics are *diverse*, and plentiful; also, the scheduler may not require knowledge of all of them at the same time. For instance, while the cache-miss ratio reported by the processor can be a helpful information for an affinity-based scheduler [12], it may be completely useless for a different scheduling strategy such as co-scheduling [51]. Finally, some of the characteristics are *dynamic*, and may accumulate at a high rate; this is caused by changes in the application workload, and by other varying system behavior. Also, due to the timeliness of scheduling decisions, much of the accumulating information can become obsolete very quickly.

1.1 The Problem

There exist several approaches to support parallel scheduling in operating systems; however, either they fulfill only part of the requirements that arise from the nature of scheduling information, or they suffer from other limitations and inflexibilities. *Pure kernel-based schedulers* have the required scheduling information right at hand. However, they usually offer only a limited set of scheduling policies [5,9,48,57]. Moreover, with a scheduler entangled with the rest of the kernel, even adjusting a scheduling policy can become a complex and daunting task. *Application-level schedulers*, such as user-level thread libraries [3], offer the desired flexibility with respect to the policies; they are free to implement an arbitrary policy, and extending a current implementation is a rather easy task, with minimal implications on the rest of the system. Nevertheless, they are inherently inflexible with respect to their *scope*, since they are bound to a specific application domain. Also, application-level schedulers may exhibit poor performance, due to the lack of coordination between user- and kernel-specific semantics [4]. *Two-level schedulers* attempt to combine the advantages of pure kernel- or application-specific schedulers [4, 47, 62]. These approaches basically employ a kernel- *and* an application-level scheduler, and a means to propagate scheduling-relevant information and events in between. These approaches generally relieve the system from drawbacks the pure variants suffer from. However, either they provide propagation mechanisms that fail in cases of fluctuating or frequently accumulating scheduling characteristics, or they focus only on conveying scheduling data, but do little to address the problem of obtaining the data. Finally, also *extensible kernels* offer the opportunity to overcome the inflexibility of kernel-based schedulers. Extensible kernels allow applications to define operating system functionality themselves, which is typically achieved by means of downloading application-specific code into the kernel [13, 22, 54]. Thus in principal, extensible kernels combine the advantage of kernel-based schedulers, which have direct access to scheduling information with the advantage of user-based schedulers, which allow for application-specific policies. However, extensible kernels suffer from a variety of hard problems related to safety, and require sophisticated countermeasures to address these problems [20].

1.2 Approach

The weaknesses of current approaches have motivated us to introduce a new mechanism which is more suitable to address the requirements of user-level schedulers in the face of distributed, diverse, and varying scheduling characteristics. In this work, we present a novel approach to support multiprocessor schedulers at user-level. We propose to use *system instrumentation* and *event logging*, in order to obtain run-time information relevant for scheduling.

Inspired by the efficiency and flexibility of performance monitoring tools, we leverage instrumentation techniques to enhance the system components with handlers that record scheduling-relevant data into designated log files held in main memory. By making sure that the instrumentation handlers are safe and well-defined, we retain applicability of our approach even to crucial system components such as the operating system kernel, allowing thereby for a unified infrastructure for collecting scheduling data. The asynchronous nature of event-logging and its broadband interface render our approach an efficient solution even in cases of fluctuating or high-volume scheduling data. In addition to the basic logging mechanisms, we provide several concepts for flexible and efficient management of our logging infrastructure. Among these concepts are a method to dynamically enable and disable logging on a per-characteristic base, and an interface to dynamically specify the scheduling data emitted to the log files. We also introduce the new concept of *accounting domains*, which allows schedulers to account scheduling data to different entities in the system. Our log infrastructure is designed to permit efficient access to log buffers, both by schedulers and the instrumented component, and to scale well with the number of processors.

To evaluate accumulation of workload characteristics via logging, we have implemented a prototype logging facility within a real-world, microkernel-based, component operating system. Our environment is based on the L4 microkernel and the L4Ka Virtualization Environment. We have successfully instrumented the microkernel, the application workload, and a user-level device driver subsystem providing services to the applications. From these components, we extract several run-time scheduling characteristics, including timing properties and inter-process communication (IPC) patterns from the microkernel, and shared-memory usage from a user-level device driver subsystem. Measurements indicate a rather undramatic baseline overhead, with zero overhead for disabled scheduling characteristics, and logging costs ranging between about 26 and 275 clock cycles on an Intel 2.2 GHz Xeon Processor. Application-level benchmarks on driver throughput indicate worst-case logging penalties of about 3.8 percent logging for IPC usage, and about 10.5 percent for logging shared memory usage. Throughput measurements were done in a scenario not bound to I/O processing, with a device driver operating on a RAM disk.

The rest of this thesis is structured as follows: Chapter 2 reviews background material and related work in the context of our approach. Chapter 3 will present and discuss the design concepts and ideas behind our system. It is followed by a detailed presentation of our prototypical implementation in Chapter 4, and an evaluation of our prototype in Chapter 5. Chapter 6 summarizes our approach and presents our conclusions.

Chapter 2

Related Work

The following chapter presents background material and related work in the context of our approach. The chapter is subdivided into three parts. In the first section, we review run-time characteristics commonly used within the context of multiprocessor scheduling. In the second section, we will discuss contemporary approaches to supporting job scheduling on multiprocessors. In the third section, we will present research done in the area of tracing and performance monitoring. Although having a different background and goal, our work coincides with such tracing and monitoring facilities in several key aspects.

2.1 Run-time characteristics used in Parallel Scheduling

Following the taxonomy introduced in [17], the general scheduling function can be defined as a mechanism that allocates processing resources to consumers, based on a scheduling policy. We can consequently define multiprocessor scheduling as a mechanism that allocates *multiple* processing resources to consumers, based on a given policy. Scheduling principally relies on information. As the problem of allocating processors to consumers, most of the measures and characteristics relevant for multiprocessor scheduling are naturally related with the processors, or the consumers, or both. Furthermore, many scheduling characteristics are relevant on a p̂er-consumer base, that is, they attribute the processor consumer with regard to upcoming scheduling decisions. Some of the deciding factors of scheduling are of static nature, such as predefined job classifications or priorities. However, a major part of the characteristics *accrue dynamically*, during run time of the system. Our work focusses on those run time characteristics; we have classified them into four major categories: (i) usage of the processor, (ii) usage of processor-associated resources, (iii) usage of shared resources, and (iv) system load indicators. In general, the first three categories cover the properties that are relevant on a per-consumer base; we therefore use term *usage* instead of characteristic. To denote the consumers of the various resources, we will use the term *resource principals* hence. In contrast, system load indicators are typically global system properties, and at most they reflect resources that have been consumed by all principals, or by none. In the following, we will review relevant scheduling policies grouped by those four categories, and detail how the policies extract the required characteristics from the run-time system.

2.1.1 Usage of the Processor

The processor is the central resource within the context of scheduling; consequently, many scheduling characteristics are related to the processor and its usage by the processor principals. Perhaps the most evident example of a processor-related characteristic is the *service time* of a principal, which is the time a principal has used a given processor. Since the service time correlates with its parallelism, timing characteristics can be considered as *parallelism* characteristics as well. Traditional queuing models simply use the cumulative service time on all processors to characterize principals; however, research has shown that such a single-value parameter is an inadequate characterization, and has therefore proposed more elaborate timing parameters such as the *processor working set*, the *execution signature*, or the *parallelism profile* [27, 55]. In addition to the actual service time, the *coefficient of variation* of the service time can help to determine, if the service time is predictable at all [28]. Although timing characteristics are often assumed to be known a priori, there have also been efforts to measure them at run time [15, 28, 50]. Such measurements principally require instrumentation of the execution points where a unit of work has been completed, such as context-switches between kernel-, or user-level threads, or, at a coarser grain, start and stop times of whole jobs or activities.

More recent efforts investigate the applicability of *processor-internal* characteristics to multiprocessor scheduling. For instance, the approach in [1] measures mean cycles-per-instruction per thread, with the goal improve scheduling on multi-threaded processors. Similarly, the approach in [29] proposes to measure the energy consumption of threads, in order to perform scheduling based on energy criteria. These approaches usually obtain the required characteristics from hardware sensors and performance monitoring counters.

2.1.2 Usage of Processor-associated Resources

When a ready run processor principal is to be scheduled in a parallel system, it may be required or beneficial to map this principal to a specific processor. One factor that leads to the increased affinity is the usage of processor-associated resources. For instance, some resource may be available only on a specific processor, and the principal is forcibly bound run on these processors. Other resources may contain some "state" while being used, and principals that make use of them in a timely or spatially local manner improve their efficiency. Finally, resources may be used concurrently by multiple principals; running together principals with high usage rates may have negative performance implications, due to contention and interference. Knowledge of usage patterns of processor-associated resources can therefore significantly improve the effectiveness of parallel job scheduling.

Perhaps the most important example of a processor-associated resource is the processor's caches. *Cache affinity scheduling* basically attempts to reduce the amount of state that principals must reload into the caches after being (re-)scheduled on a processor. [58]. Cache affinity scheduling can significantly improve performance if context switching rates are high [12, 61], or if the costs of reloading cache state are substantial, like on on machines with high memory latencies [18]. Several techniques are used to reduce the amount of cache state: one scheme is to re-schedule a principal on the processor it last used, since there may still be "hot" state in this processor's cache from the last run. Another approach is to let a processor favor those principals, which have run on the processor in the near past [18]. The last scheme tries to reduce the cache

interference by scheduling principals with a high mutual cache locality in consecutive order [11]. Cache affinity policies employ different schemes to measure the cache state of a processor principal: some approaches simply model the cache state based on the service time of the principal [9, 18]. Alternative approaches leverage hardware support to obtain the number of cache-misses per principals [12, 65], or equivalent information from the processor's translation look-aside buffers [11].

Besides the cache state, also the *memory requirements* of a principal play an important role in scheduling. Scheduling policies make use of memory considerations in basically two ways: first, on machines with non-uniform accesses to memory, some approaches preferably run the principals on those processors or nodes, which host their memory working set; cache-misses are then being mainly serviced from local rather than from remote memory [18, 70]. Second, some approaches attempt to reduce memory pressure, either by using memory-based admission controls [10], or by preferring principals in proportion to their memory requirements [52]. To obtain the memory allocations to principals at run-time, memory-conscious scheduling policies typically query the memory management subsystem, which is responsible for allocating memory pages to the principals.

2.1.3 Usage of Shared Resources

The idea of scheduling cooperating processor principals together is widely referred to as *co-scheduling*; it was initially proposed by Ousterhout [51]. Co-scheduling assumes that parallel programs have a *process working set*, which must be scheduled simultaneously for the program to make progress. A technique that is very similar to co-scheduling is *gang scheduling*; the subtle difference between those variants is that gang scheduling only schedules complete working sets, whereas co-scheduling resorts to a subset, in case that there are not enough processors available. Co-scheduling enhances the flexibility of scheduling, but breaks the performance guarantees gang-scheduling can give.

Ousterhout originally relied only on statically specified input by the application [51]. However, more recent approaches show, that run-time identification of process working sets is a possible and efficient alternative [25, 56, 65]. Run-Time identification is principally based on monitoring usage of *shared resources*. Similar to usage of processor-associated resources, which increases the affinity to a processor, usage of shared resources increases the affinity of principals to each other. A scheduling policy that is aware of the interaction patterns between principals can use its knowledge and schedule cooperating principals together, in order minimize synchronization times and increase overall performance. In practice, different techniques exist to monitor usage of shared resources: the approach in [25] identifies processor working sets by instrumenting communication channels. This approach requires that the channels are identifiable, which holds true, for instance, for named communication facilities, or synchronization locks. However, if communication is based on shared memory, the approach will fail, since this form of communication is more opaque and harder to detect. The approach in [56] monitors shared memory interaction indirectly, by periodically scanning the contents of the processor's translation look-aside buffers. Finally, the approach in [65] proposes to obtain sharing patterns from hints, which are specified by the applications themselves.

2.1.4 System Load Indicators

The last category of scheduling characteristics covers all properties used to measure the load of the system. In contrast to the characteristics that fall into the other categories, system load indicators are typically global system properties. At most they reflect resources that have been consumed by all principals, or by none. The current load situation is principally a run-time system property. It is taken into account in order to keep the fluctuating workloads balanced among the available processors. The most widely used load indicator is the length of the processor run-queues, as for instance used in [8] and [53]. But other single-value load descriptors are possible, such as system call or context switching rates, or processor idle times [39].

2.2 Approaches to Support Parallel Scheduling

In this section, we will review current approaches to support parallel job scheduling on multiprocessors. We begin with the two classical schemes used in operating systems, kernel-level and application-level scheduling, and show that those approaches are inherently limited with respect to flexibility and scope. We then review several advanced approaches to supporting multiprocessor scheduling. Although these approaches provide a more flexible for scheduling, they still suffer from efficiency and safety problems.

2.2.1 Kernel-Level Schedulers

Virtually all modern multiprocessor operating systems, including UNIX- and Windows-based systems, export a notion of election context to the user – e.g, thread, process, or virtual machine. The operating system multiplexes execution contexts among the physical processors by means of a scheduling subsystem. The kernel-level scheduler subsystem deals both with collecting required scheduling data, and dispatching concurrent context among the available processors, based on a given scheduler policy. The scheduler runs at privileged mode, and dispatches only visible, first-class execution contexts. Scheduling characteristics are cheap to accumulate, since hardware properties and other kernel subsystems are directly accessible, without the need to cross protection boundaries.

However, kernel-controlled scheduling suffers from a number of problems and limitations. First, kernel schedulers often export execution contexts that are tied to other operating system abstractions, such as protection domains; this limits flexibility and may lead to high overhead for switching those contexts [3, 7]. Second, although most operating systems support more than one scheduling policy [5, 9, 48, 57], and some even accept scheduling hints from users [14], kernel schedulers are inherently limited with respect to the supported scheduling policies. Since the scheduler subsystem is embedded and entangled with the kernel, even adjusting an existing scheduling policy can become a complex and daunting task.

2.2.2 Application-Level Schedulers

The limitations of kernel schedulers have led to an emergence of user-level thread packages as a new abstraction for supporting application parallelism. Such libraries multiplex a set of user-controlled threads among a single, kernel-exported execution context.

An application-specific scheduler is responsible for dispatching user threads among the kernel context. Scheduler and threads reside within the same protection domain, and the required scheduling information is – as it is the case for kernel-level schedulers – right at hand. Aside from being faster as kernel threads [3], user-level threads greatly enhance flexibility of scheduling. Thread libraries are free to implement any desired policy, and extending current implementations does not have any side-effects on the rest of the system. However, these benefits come at a cost. First, thread packages may exhibit poor performance, since the underlying kernel-level scheduler is oblivious to the application-specific notion of threads [4]. Second, although thread packages provide the required flexibility with respect to the policy, they are still inherently inflexible in their *scope*. Thread packages are bound to a specific protection domain, thus cannot be used as a foundation for development of a scheduler that manages multiple applications or protection domains.

2.2.3 Two-Level Schedulers

Since neither kernel-threads nor user-threads are fully satisfying concepts, research has investigated, how the benefits of both solutions could be combined within a single approach to express and manage parallelism. Such combinations are widely referred to as two-level schedulers. In the following, we review three such approaches: Process control, Scheduler Activations, and First-Class User Level Threads.

Process Control

The idea of process control is based on the observation that performance degrades when the total number of processes exceeds the number of available processors. Process control is a mechanism that allows parallel applications to dynamically adapt the number of execution contexts, in order to achieve a one-to-one mapping from execution contexts to real processors [62]. For this purpose, process control employs a centralized scheduler, which periodically polls the kernel for the number of runnable processes; in case the number does not match the number of processors, it requests applications to suspend some of their processes. For safety reasons, the applications must suspend and resume their respective processes themselves. In contrast to pure user-level threads, process control allows for scheduling of multiple applications and protection domains. Also, it does not suffer from the limited extensibility kernel-level schedulers are subject to. However, it introduces a number of disadvantages. First, polling is an efficient approach to promote scheduling characteristics *only* if the frequency of alterations is sufficiently constant; process control is thus limited to policies that do not require characteristics accruing with varying frequencies. Second, process control relies on cooperative behavior of the applications, since there is no means to provide the scheduler with scheduling information required to determine if a process can be suspended safely.

Scheduler Activations

The concept of scheduler activations is driven by the insight, that kernel-threads are the wrong base for user-level thread packages. For this reason, scheduler activations are introduced, as a new foundation to user-level management of parallelism [4]. In this approach, a kernel scheduler allocates complete processors to address spaces, rather than only processes; it hands over the dispatching of threads to a user-level scheduling

system, by reflecting all relevant scheduling events (e.g., the blocking of a user-thread within the kernel), to the user-level scheduler. This is achieved by means of a notification mechanism, which is dubbed scheduler activation. The activated user-level scheduler can then respond to the change according to its own requirements. Like process control, scheduler activations overcome the limitations of both application-specific and kernel-specific execution contexts. However, propagating scheduling information via synchronous upcalls suffers from similar problems as the polling mechanism used in process control. Scheduler activations are inherently limited to events with a comparatively low frequency. Since a scheduler activation requires the switching to user-mode, notification incurs heavy performance costs, if the frequency of events is high. Also, there exist scheduler-relevant events (e.g., cache-misses) that do not require invocation of the scheduler the instant they take place, rather, it is sufficient to *record* the particular event for future use by the scheduler.

First-Class User-Level Threads

First class user-level threads is an approach similar to the two previous ones in that it attempts to bring user-level and kernel-level abstractions closer together. The approach grants user-level threads a first-class status: the kernel remains in charge of coarse-grain thread management, while thread packages implement the fine-grain functionality that does not require involvement of the kernel [47]. Like with scheduler activations, the kernel notifies the thread packages on pending scheduling decisions, by means of a software interrupt. In addition, kernel and thread packages convey important thread-related data via shared memory, that is, without the requirement for synchronous interaction. Thus, first class user-level threads are generally capable of communicating scheduling-related information between user and kernel. However, their approach focusses on exchanging data, and only on exchange between kernel and user; it does little to address the problem how data *accumulation* can be done efficiently. Also, there is no support for data propagation *between* user-level components.

2.2.4 Extensible Kernels

Extensible kernels attempt to overcome the inefficiency and inflexibility of poorly matching operating system services and interfaces. The basic approach is to let applications define the functionality constituting the operating system themselves, by allowing them to download application-specific code into the kernel. Research on extensible kernels has proposed several ways to manage job scheduling at user level. SPIN lets applications provide their own thread packages and scheduler, executed as safe extension within the kernel [13]. Exokernel relies on a user-specified interrupt handler for general purpose context-switching [22]. The approach in [19] further investigates multiprocessing on Exokernel; however, this approach implements only a basic multiprocessor scheduler, and lacks support for more sophisticated than a basic load-balancing algorithm. Thus, in theory, extensible kernels overcome the existing limits of multiprocessor scheduling with respect to flexibility and extensibility. Allowing for user-defined specializations of an operating system, they can accommodate the different demands of scheduling policies, including the demands of distributed, diverse, and dynamic scheduling characteristics. However, aside from lacking evidence with respect to multiprocessor scheduling, extensible operating systems suffer from a variety of hard problems related to safety. To prevent downloading compromising or malicious code into the kernel, such systems require sophisticated countermeasures such as

run-time safety-checks, or language restrictions to a Turing-incomplete subset [20].

2.3 Tracing and Performance Monitoring Tools

Understanding the behavior of an operating system is a challenging task, and tracing infrastructures can provide valuable aid in debugging, analyzing or tuning operating systems. Examples of such infrastructures are the Digital Continuous Profiling Infrastructure of DEC [2], the tracing infrastructure in K42 [66], DTrace of Sun Microsystems, [16], or the Linux Trace Toolkit [68].

These infrastructures feature many techniques and mechanisms that are applicable to our approach as well, such as dynamic instrumentation [59] or lock-less event logging [66]. However, tracing facilities differ in two key aspects, which prevented us from directly incorporating such a tool to solve the problems of accumulating scheduling data. The first main difference lies in data lifetime. Tracing data is often preserved for a long time (e.g., in a disk database [2]), for later analysis by humans or graphical tools. Since this may take place off-line, tracing data can generally become very large in size. In contrast, scheduling characteristics serve as a means to predict the future *online*, based on the near past. Due to the frequency of scheduling decisions, the collected samples usually become stale very quickly. Also, preserving the samples in a database is not generally required. The second key aspect is that the debugging and tracing tools trade off the three criterions efficiency, flexibility, and safety in a way that we deemed unacceptable for our purpose: DEC's Digital Continuous Profiling infrastructure allows for efficient profiling, but is restricted to a fixed set of trace events. K42's tracing infrastructure also allows for efficient data accumulation, but was not specifically designed with safety in mind. Sun's DTrace features the ability to dynamically specify trace events and handlers at run-time; designed for a production environment, DTrace also assures that the execution of event handlers is absolutely safe. However, it does so by executing probe handlers in a safe environment, and thus suffers from the same complexity problems as other extensible kernels. Finally, the Linux Trace Toolkit only focusses for kernel tracing, and provides no support for application-specific tracing.

2.4 Summary

To summarize, run-time characteristics relevant job scheduling fall into four major categories: (i) usage of the processor, (ii) usage of processor-associated resources, (iii) usage of shared resources, and (iv) system load indicators. All these characteristics arise from different sources in the system, and most of them are relevant only for specific scheduling policies or environments. Some of them are slow-paced, but others are highly fluctuating parameters and accumulate very frequently. Previous approaches to support parallel either suffer from deficiencies with respect to scheduling information, or they are overly limited in other respects:

- Pure kernel- or application-level approaches offer limited flexibility and extensibility, supporting either only a fixed set of policies, or restricting the scope to a single application or protection domain.

- Process control and scheduler activations achieve the desired flexibility and extensibility, but may exhibit poor performance in cases where scheduler-relevant properties change very frequently. First-Class User-Level threads overcome the

efficiency problems of scheduler activations and process control, but are focused on interaction between the kernel and user-level components, and do little to convey scheduling events between the user components.

- Extensible kernels theoretically meet the demands of scheduling characteristics with respect to flexibility and efficiency, but introduce safety and security related problems.

Finally, while tracing and performance monitoring tools provide many features and concepts that are applicable to the context of scheduling as well, they are not directly usable for that purpose; specifically the different trade-offs with respect to efficiency, flexibility, and safety renders a one to one adoption a questionable approach.

Chapter 3

Design

In this chapter, we will present the design of our approach to support user-level scheduling via instrumentation and event-logging. We first point out a set of requirements and goals that should apply to our solution. In the subsequent sections, we will present our proposed approach, and then detail how it achieves the desired requirements and goals.

3.1 Requirements and Goals

We begin by defining the requirements and goals the design of our approach should reflect. In particular, we want our solution to be:

- **Flexible, and unified**. If the scheduling subsystem is not limited to a set of policies – such as non-extensible kernel schedulers –, or to a specific application – such as user-level thread packages –, the scheduling data is inherently diverse, and distributed among the whole system. A valid solution must therefore be suitable for different components, not just for specific subsystems or applications.
- **Applicable to the kernel**. As a major source for scheduling-relevant information, the approach must be apt also for use within the most privileged and critical part of the operating system. However, we cannot permit arbitrary extensions of kernel code or interfaces, since they compromise the whole system's safety [20].
- **Efficient**. To meet the demands of real-world environments and dynamic workloads, the solution must accommodate the inherent fluctuations in scheduling characteristics and the resulting abundance of scheduling data. Obtaining, gathering, and delivering scheduling information needs to be cheap, and fast. Solutions based on synchronous notification [4], or on polling [62], will fail in the cases where changes in scheduling-relevant characteristics occur too frequently.
- **Scalable**. Naturally, support for multiprocessor scheduling must scale with the number of processors. In this work, we focus on shared-memory multiprocessors with uniform or non-uniform access to memory.

3.2 Proposed Scheme

This work presents a novel approach to support multiprocessor scheduler extensions at user-level. We propose to use system instrumentation and event logging to obtain

run-time information relevant for scheduling from different system components and to store this information within designated log files. To deliver the accumulated scheduling information, we share the log files with the user-level scheduler application. The scheduler can finally analyze the shared log files to base its allocation policies on. Our approach is illustrated by Figure 3.1. There are several concepts and aspects to our solution:

Efficient and safe logging and instrumentation. By making sure that the log handlers are cheap and small, we achieve that the data can be accumulated and delivered very efficiently. We further reduce the overhead by allowing the scheduler extension to dynamically enable and disable instrumentation on a per-characteristic base. However, for safety reasons, we deploy only well-known and predefined instrumentation code, to retain the applicability of our approach even to safety-critical environments such as the kernel.

Flexible log control and data specification. To allow for flexible data accumulation, we provide an interface that allows schedulers to specify and control the data flow through the log infrastructure. However, rather than to allow the specification of arbitrary log contents and layouts, the interface only offers to choose from a predefined set of possibilities. Limiting the degree of freedom not only retains good performance while still being flexible enough within the context of scheduling; it also enables us to provide a safe and well-known interface to the scheduler which never results in undefined behavior.

Flexible log data accounting. The diversity of scheduling policies often comes along with inherent ambiguities in the definition of schedulable or accountable entities. To comprise all different types and granularities of entities that may exist throughout the system, we introduce the new notion of an *accounting domain*. For the logging infrastructure, an accounting domain is merely a pool of resource principals, and a set of dedicated log buffers that can be used to record scheduling information in. The management of associations between resource principals and accounting domains is exported to the scheduler application. This concept not only enhances flexibility, by handing over the policy-associated definition of accounted entities to the scheduler. It also serves as a performance improvement, since pooling principals into a domain implies, that interactions *within* the pool are not relevant for the scheduler, and can be filtered out by the logging facility.

Efficient log data aggregation. The log infrastructure provides several mechanisms to aggregate and reduce the log data. Schedulers can configure and use these mechanisms to aggregate the volume of accumulated data depending on their own demands and requirements. We provide the mechanisms not only only to meet performance requirements; aggregation techniques also make our facility more flexible, since they allow the data flow to be adjusted to reveal specific questions.

Efficient and Scalable Log Data Access. Rather than to incorporate copy-out or double-buffering techniques, we share the log buffers directly with the user-level scheduler. Log buffers can be analyzed the instant the scheduler requires knowledge of the information. Using lock-free synchronization between the log data producers and consumers, we further make sure that data can be accumulated efficiently, without requiring costly inter-processor synchronization operations. For reasons

of scalability, log data is recorded in per-processor buffers; producers of log data on different processors operate completely independent from each other.

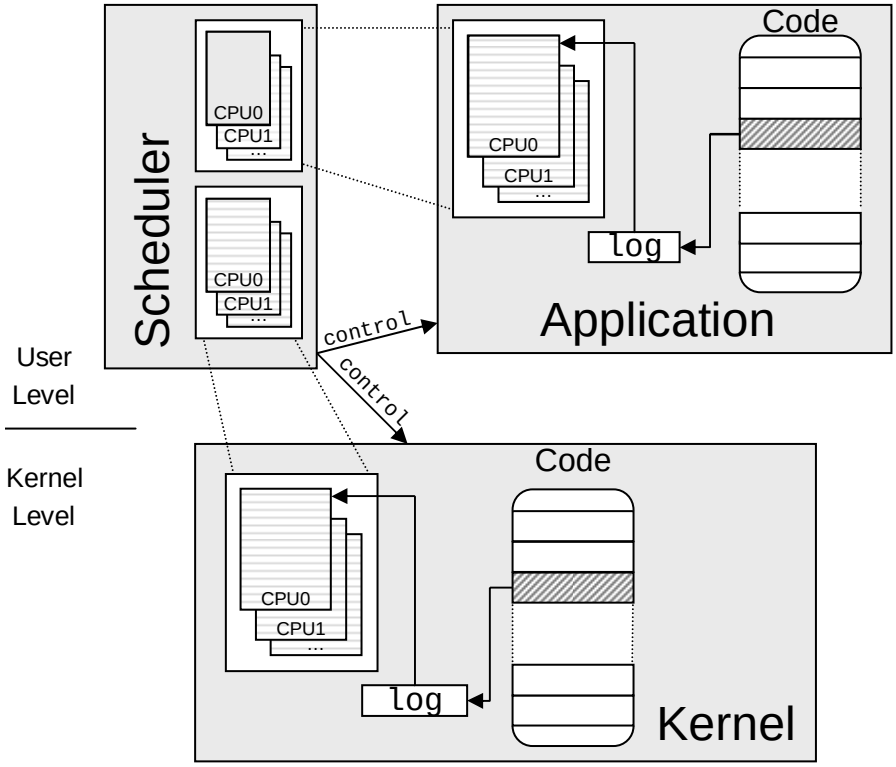

Figure 3.1: Design overview. The log infrastructure can instrument the kernel and user-level modules or applications. At the respective instrumentation points, scheduling characteristics are logged in per-processor log buffers. The log buffers are shared with the scheduler application, which bases its decisions on accumulated information. Additionally, the facility exports a control interface to the scheduler, which allows for flexible log control.

In the remainder of this chapter, we will discuss the previous aspects in more detail. We will first discuss the concepts behind logging and instrumentation (Section 3.3). We afterwards discuss issues related to log control and data specification (Section 3.4), and to the accounting of log data (Section 3.5). We continue by describing the data aggregation mechanisms (Section 3.6). In the final section, we discuss issues and considerations related to log data access (Section 3.7).

3.3 Logging and Instrumentation

In this section, we discuss the basic concepts that a logging and instrumentation facility suitable for scheduling-relevant information should implement. In general, these concepts must meet two main design requirements: efficiency and safety. More specifically, while logging must be efficient enough that the costs of accumulating scheduling

data do not outweigh its benefits, it still must be safe enough to be applicable also to the privileged operating systems kernel.

Hollinwgsworth et al. describe instrumentation as: "the activity of collecting information about an execution without modifying the intent of the underlying calculation" [35]. This is basically accomplished by enhancing the system with probe handlers or event hooks, that calculate and record the requested information. Using instrumentation for the purpose of *logging* consequently means to define a set of appropriate instrumentation events, and to enhance these events with a *log mechanism*. This mechanism is activated whenever control flow reaches the instrumentation events, and records the occurrence of the event – together with other useful information – into a designated log file. Instrumentation and event-logging are commonly used technique in the context of debugging, profiling, and performance monitoring [16, 46, 49, 59, 66]. In contrast, we propose to use instrumentation to collect *scheduling-related* information from the diverse system components. However, although the mechanisms are principally the same, we must consider the special requirements when using instrumentation to accumulate scheduling data: first, due to the timing constraints of scheduling-relevant data, instrumentation should be as efficient as possible. Second, since we want to apply this technique to the kernel as well, instrumentation must not have any implications on the system's safety.

Many performance analysis and debugging tools allow users (e.g., programmers, compilers, or applications) to define themselves the actions that should be taken whenever an instrumentation point is reached. This is typically achieved by means of a description language. The description language constitutes a dynamic interface that is used for translating user-specified semantics into machine code. The code is then injected into the instrumented component sometimes at compile-time [46], but preferably at run-time [16,35]. Further, many of the dynamic instrumentation systems are capable of instrumenting almost any kernel and user-level code [16,35,59]. Dynamic interfaces and instrumentation greatly enhance the flexibility of these tracing tools, allowing for instance the definition of arbitrary actions at run-time and code injection into unmodifiable components and modules. Dynamic instrumentation techniques also improve efficiency, since they allow for enabling and disabling the functionality at run-time. However, these features come at a cost. While dynamic extensions and interfaces may be well-suited and unproblematic for user programs, applying these techniques to the operating system kernel – at least at arbitrary locations, or with arbitrary application-provided code –, can clearly jeopardize the whole system safety. To overcome the problems, these techniques must incorporate sophisticated countermeasures like run-time safety-checks, or language restrictions to a Turing-incomplete subset [16, 35]. Particularly for small or safety-critical system components, with a concise and well-defined interface (e.g., microkernels, or virtual machine hypervisors), neither complex safety-checks, nor mutating kernel interfaces are a tolerable solution.

To retain some of the flexibility and efficiency of such dynamic techniques, but to ensure on the other hand, that instrumentation and control is safe even within the kernel, our approach makes a compromise between dynamic and static techniques. It is based on the insight that, although scheduling characteristics are inherently diverse and distributed, they still show structural and semantical similarities. This basically allows us to use only predefined instrumentation events and handlers. By offering the ability to dynamically enable instrumentation on a per-characteristic base, we meet our safety goals, but still leave enough room for flexible and efficient customization of the logging infrastructure. In the following paragraphs, we discuss our approach in more detail. We first define several types of events that are appropriate for recording scheduling-relevant

characteristics. We then discuss the issues related with the instrumentation of these log events.

3.3.1 Log Events

Although scheduling information is inherently diverse, and distributed among different system components, the characteristics still show structural and semantical similarities. This mainly stems from the fact that scheduling is a processor allocation problem, which results in a focus of the characteristics on processors and processor-related resources. As discussed previously, run-time scheduling characteristics generally fall into four different categories: usage of the processor, of processor-associated, or of shared resources, and, finally, system load indicators. In light of these categories, we have identified three types of events that are appropriate for logging scheduling-relevant workload characteristics: (i) events where a characteristic changes or a resource is accessed, (ii) events where a principal enters or exits a phase of *continuous* access to a resource, and (iii) asynchronous, unrelated events. The following paragraphs elaborate on these three types of events in more detail.

On-Change/On-Access Events

On-change and on-access events are the most intuitive type of a scheduler-relevant event. They simply denote an event, where a certain property changes, or resource access takes place. An example of such an event is the access to a shared software object by a principal. Another example would be the insertion of threads into a processor's run-queue.

Not all on-change or on-access events can be instrumented in practice. Only visible software events, represented by a code path, can be enhanced with an appropriate code construct. This may not be the case with properties maintained by hardware, or with properties hidden in unmodifiable or closed, binary-only components. Furthermore, such events can only be instrumented efficiently, if changes or accesses occur not too frequently. Otherwise, instrumentation of every access will incur unacceptable overhead.

On-Entry/On-Exit Events

Extracting scheduling-relevant characteristics on access or on change is a simple, but often not applicable scheme. However, for resource usage by principals, which forms a major part of the scheduling-relevant characteristics, one can sometimes use a different set of events instead: based on the assumption, that usage of a resource is exclusive to one principal for a certain time, one can instrument the entry and the exit of such a phase of resource access. We refer to such a pair of events as *on-entry* and *on-exit* events. The alteration in the resource usage can afterwards be charged to the respective principal, without the requirement of tracking each single access. Alternatively, if the resource access is known to be continuous, the time difference can be used to approximate the actual alteration.

An example where such a scheme can be applied is the usage of a processor by a thread. The difference in the number of cycles – or alternatively, in time – between a thread's dispatching and preemption event yields the actual processor usage count of the thread (See Figure 3.2). Another practical application for this scheme would be a client-server model, with communication based on shared resources. To track

the "usage" of the server by client principals, one can instrument the event where a client calls the server for service, and the event where the server completes this service request. The communication intensity, counted for instance in the number of calls to communication routines, or in the number of references to shared memory pages, is afterwards charged to the client principal.

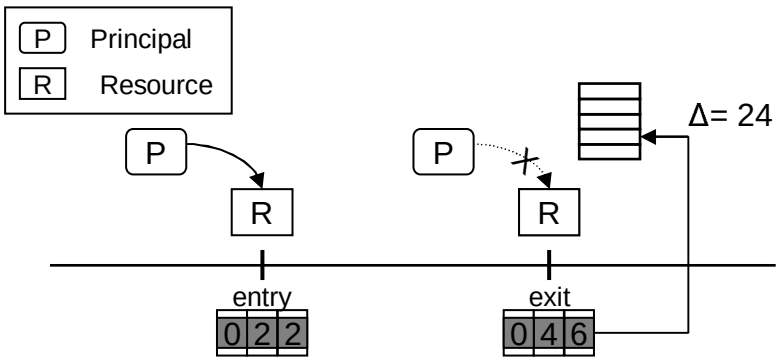

Figure 3.2: Entry/Exit event type. Based on the assumption, that usage of a resource is exclusive to one principal for a certain time, one can instrument the entry and the exit of such a phase of resource accesses, and charge the alteration to the respective principal.

Asynchronous Events

Both previous event types, on-change/on-access, and on-entry/on-exit events, are synchronous to the actual change of the respective characteristic. The third class of events covers all *asynchronous* events, which are unrelated to the actual alteration. Commonly used unrelated events are periodically invoked software functions, such as the timer interrupt handler. Alternatively to periodic events, also random events can be instrumented, to provide a set of samples of a certain scheduling characteristic.

Recording a characteristic by instrumenting unrelated events assumes that the information has already accrued somewhere in the system. Furthermore, instrumenting unrelated events may not result in a complete record of characteristics, since alterations between the events will not emerge into the record. However, it is more efficient in cases, where a less detailed record of scheduling characteristics provides enough information to the scheduler, and a synchronous instrumentation is thus not justified.

3.3.2 Component Instrumentation

To measure and extract the desired scheduling characteristics, our infrastructure inserts instrumentation handlers at the respective log events. By designating different instrumentation points for different scheduling characteristics, we can ensure that accumulation and logging is always performed in the same way at each point. We are therefore able to only insert well-defined instrumentation handlers. Each handler is basically a small log mechanism, that accumulates the desired data and writes it out into an assigned log file. Using only instrumentation code that is known at compile-time not only ensures safety of the instrumentation process; it also allows us to give

performance guarantees on the whole logging process, by using only time-bound instrumentation code.

However, the diversity of scheduling characteristics, and the fact that an actual scheduling policy may require only a small subset of the accruing characteristics, has convinced us that accumulation of all possible scheduling data at the same time is clearly an inefficient and intolerable solution. We therefore make use of *dynamic* instrumentation techniques, and allow schedulers to enable and disable logging at runtime. More concrete, all logging code enhancements are removed by default. Upon request by the scheduler, we enable the logging of a given characteristic, by dynamically inserting the respective code enhancement into the instrumentation points corresponding to the characteristic. Using dynamic code insertion permits us to enable logging only for those characteristics that are really required by the scheduler. It also eliminates the performance penalties of disabled-probe checks that would otherwise burden the original code path.

Inserting code fragments at arbitrary locations requires mechanisms such as trampolines [35], or splices [59], to safely patch the respective location, and to relocate and later execute the overwritten original instructions. In contrast, the limitation to a set of well-known instrumentation points allows us to use a much simpler approach: we prepare all log events with a sequence of place holders (e.g., `no-op` instructions), that make room for later use by the instrumentation. Upon request, we simply insert the respective log code enhancement into the space previously reserved by the place holders. For further details on dynamic instrumentation, we refer to our implementation, which is presented in Chapter 4.

3.4 Log Control and Data Specification

Our approach intentionally limits the control flow induced by instrumentation, for safety and efficiency reasons. All inserted handlers consist of predefined code fragments known at compile time. Using dynamic instrumentation, logging can still be enabled and disabled during run time. We use a similar scheme for the data flow, that is, the generated log data produced by the logging code enhancements. The similarities in the semantics of scheduling data enable us to again limit the degree of freedom to a set of predefined log layouts and contents, still providing however a flexible enough mechanism for managing the log data flow. The scheme is illustrated by Figure 3.3, and works as follows: Each logging component exports a well-defined interface, which allows schedulers to route scheduling data from different characteristics to *separate* log files. Further, this interface allows schedulers to choose the *contents* of each designated log file, by selecting for each a subset of the following predefined log entries: (i) event identifier, (ii) resource principal identifier, (iii) counter, (iv) correspondent principal identifier, and (v) time stamp. In the following, we describe each of the possible entries in more detail.

Event Identifier. If more than one scheduling characteristic is logged in the same log file, the event identifier is used to distinguish data from different sources. Event identifiers are not *true* identifiers in the sense that they uniquely denote the event. Rather, they uniquely denote the scheduling characteristic the event corresponds to. That is, in the case of on-entry/on-exit events, both events will map to the same identifier.

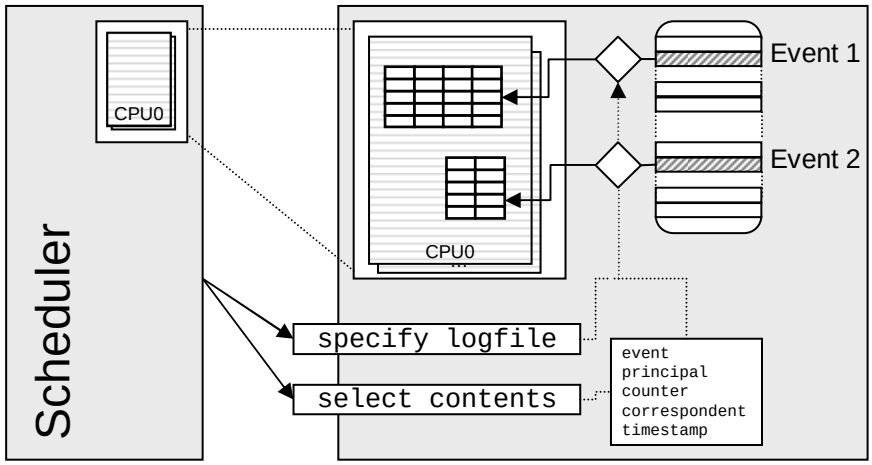

Figure 3.3: Flexible but safe log control using separate log buffers and predefined log entries.

Resource Principal Identifier. If an event reflects the usage of or access to a scheduling-relevant resource, the principal entry is used to record the particular resource principal. Such a principal may for instance be an address-space that allocates a memory page, or a thread that causes a miss in the processor cache. Similar to the event identifiers, the principal identifiers do not uniquely denote the resource principal, but rather map to the principal's *accounting domain* (See Section 3.5 for details on log data accounting).

Counter. This variable stores the actual alteration, or access counter associated with the particular event. If the event is of type on-access or on-change, the counter is hardwired to 1. Otherwise, in case of on-entry/on-exit-events, the counter is used to log the "delta" of resource accesses at the exit event. For asynchronous events, the counter value reflects the current state of the particular scheduling characteristic, such as the length of the run-queue, or the number of memory allocations to a principal.

Correspondent Principal Identifier. Access to shared resources is captured by logging the correspondent principal at each resource access. For instance, when logging access to a communication facility, the correspondent entry holds the communication partner of the current principal. For resources shared among multiple principals (e.g., broadcast or multicast communication), each correspondent must be logged separately, using a new log entry row.

Time stamp. The time stamp stores the current time at which logging has taken place. Logging the current time is useful, for instance, to identify data that has already become obsolete at the time the scheduler makes its decision. Depending on the instrumented component and the underlying hardware, we offer the selection of different types, such the hardware time stamp counter reporting the number of clock cycles since boot, or a logical time stamp counter, which is merely a variable that is incremented after being read. Of course, alternative sources are also possible, such as the real-time or BIOS clock.

There are several benefits to offering only a set of fixed but selectable log layouts: first, it enables us to provide a clean and safe control interface to the scheduler application that can be as simple as a set of flags determining if a specific log entry should be written or not. Second, we can assume that log entry sizes are fixed, by requiring the scheduler to configure the interface appropriately. Constant log entry sizes allow for both simpler and more efficient logging and reading out, as the starting point of a log entry is always known [66]. Third, and last, we can incorporate a set of in-place aggregation mechanisms, since we can assume well-known semantics of log entries (Details on log data aggregation are presented in Section 3.6).

While some of the log entries are implicitly available at the time a log operation is performed, others must be explicitly accumulated by the log mechanism. The step of accumulation is dependent on the particular workload characteristic. Data maintained by hardware is typically read out using a special processor instruction. Software properties such as the run-queue length require data structures to be parsed, or interfaces to be queried. In most cases, the information is either internal to the instrumented component, or can be obtained via an existing interface. However, a special case occurs, if only the log event is internal to a component, but the information is located at a different place, and no query interface exists that could be used obtain it from there. As an example, page *usage* is often monitored in a software-managed data structure located in the kernel (e.g., in a page table). However, detailed knowledge on page *sharing* is usually available within the memory subsystem, which may not necessarily be incorporated within the kernel [6, 34]. We propose to use recursive logging in this case, that is, to collect the scheduling information in a preliminary log file before promoting the complete set to the scheduler. In the example above, the memory subsystem and the kernel can share a log file, which the kernel uses to report page usage in. The memory subsystem can search the log file for accesses to shared pages, and afterwards propagate the complete information to the scheduler.

3.5 Log Data Accounting

Resource usage forms a major part of the scheduling-relevant characteristics. If not measured for load estimation, where cumulative resource demands or idle capacities are relevant as well, the usage must be charged to some accountable entity. The resource consumption is evaluated by schedulers when allocating processors to the entities (or, more precisely, to the *schedulable objects* these entities consist of). Commonly used notions of such accountable entities are threads for processor-related resources, or address-spaces for memory resources. But accounting can just as well be based on coarser-grain, or compound entities, such as applications or computer users.

In general, our logging infrastructure already allows for accounting of resource usage: whenever a resource is accessed, the log mechanism can record the respective resource principal in the designated log entry. A scheduler that later analyzes the log file is thus able to trace back each resource access to its originating principal. However, we still must address the problem, how the resource principals should relate to the entities *accounted* for the consumption. One solution would be to introduce a static mapping between resource principals and accounted entities. This is done in many general-purpose operating systems, where accounting is usually based solely on protection domains (i.e., processes). However, previous research has shown that such a coincidence poses substantial constraints on resource management [7]. Moreover, with regard to scheduling information, stipulating the notion of accounted entities does not

only have implications on flexibility, but also on efficiency: Consider for example a client-server based system with many thousands of threads and a high context-switch rate. In this case, using threads as the base for accounting will significantly decrease performance, since context-switches must be instrumented in order to monitor the per-thread usage of the processor. Moreover, if the scheduler is actually aware of which threads belong to which accountable entity (e.g., job, application, or user), such a fine-grain accounting not only affects performance, but also generates unnecessarily detailed log data.

To address this problem, we introduce a new abstraction that can be used for flexible accounting of resource usage. Our approach is driven by two insights: first, there is no unique set of resource principals in a system. Rather, this notion varies among the components and modules the system consists of. While the kernel may only provide base abstractions, like threads, or address-spaces, higher-level components may operate on a completely different level, dealing with programs, or computer users. Second, it should be the responsibility of the scheduler subsystem, not of the components, to define the entities that accounting of scheduling data is based upon. We therefore introduce the new concept of *accounting domains*, as a notion of an accounted entity. For the log mechanism itself, an accounting domain is merely an association between a resource principal and identifier emitted to the log files. The task of *establishing* these associations is exported to the scheduler extensions, which allows them to freely define the accounted entities their policies are based upon. The accounting domain concept is illustrated by Figure 3.4; we will discuss this concept in more detail in the next two paragraphs.

3.5.1 Accounting Domains and Logging

On request by the scheduler, the log mechanism logs an identifier of the principal originating the access to a given resource. The particular principal is tied to the resource, but also depends on the component the resource access takes place. For instance, although threads are the basic abstraction for consumers of processor time, instrumenting a high-level OS component such as a program loader may yield processor time based on a completely different abstraction – in this case, based on programs.

Accounting domains introduce a level of indirection between the actual principal, and the identifier recorded in the log files. The additional indirection level allows for dynamic associations between resource principals, and the accounted entities reported to the scheduler. Defining the original principal is left to the instrumented component; the accounting domain concept only provides a mechanism to *map* each principal to its corresponding accounting domain. The logging procedure uses this mapping to resolve a principal's accounting domain, and then uses the domain identifier as the principal's identifier in the log file. Likewise, when logging access to shared resources, it records the correspondent principal's domain identifier in the respective entry. Since the resolving procedure heavily depends on the particular component and resource, we leave its semantics open to the implementation. Details on how it can be realized for different environments will be presented in the implementation (Chapter 4).

3.5.2 Accounting Domain Management

Accounting domains decouple the original principal from the accounted entity the resource access is charged to in the log files. Since resource usage is accounted for

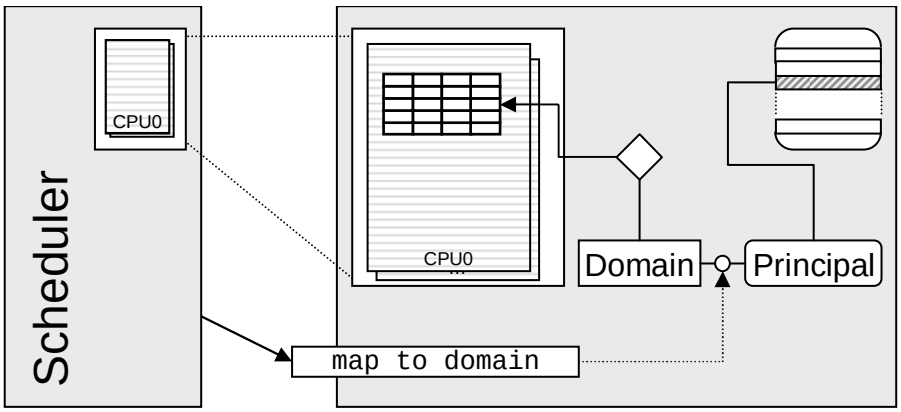

Figure 3.4: The accounting domain concept. Schedulers can define associations between resource principals and accounting domains. When logging resource usage, the log mechanism maps the originating principal to its corresponding accounting domain.

scheduling purposes, we hand over the task of defining the associations between principals and domains to the scheduler application. Each instrumented component provides a management interface that allows schedulers to dynamically bind a resource principal to an accounting domain. The log mechanism then charges all resources consumed by this principal to its accounting domain. Conversely, an accounting domain is implicitly charged for all resources consumed by its associated principals.

Based on their own knowledge of the activities in the system, schedulers can use the accounting domain concept to customize how the logging infrastructure accounts resource consumption. Similar to the concept of resource containers [7], the accounting domain concept allows schedulers to associate scheduling information with activities or users, rather than with a single resource principal. However, the accounting domain concept assumes that the scheduler is aware of the original notions of resource principals that exist in the instrumented components. For principals that are internal to a specific component, or for virtualized principals (such as virtual threads [21]), this may require additional mechanisms to establish an agreement on principals between the scheduler and the instrumented components. Further, while we allow schedulers to dynamically create and change the bindings between domains and principals, we do not provide a mechanism to modify the domain bindings within the components. Future work has to be done to investigate if such a model of “migrating principals” might be useful in the context of our work.

Until now, we have discussed the concept of accounting domains only with regard to the flexibility they add to our infrastructure. However, we introduced this concept also for a second reason: to enhance the efficiency of the system. As a pool of resource principals the scheduler considers to be “equal”, an accounting domain also enables us to group data that belongs together, or to sort out domain-internal data that is irrelevant for scheduling. These and other techniques will be presented in the following section on log data aggregation.

3.6 Log Data Aggregation

One of the main problems of run-time characteristics is their dynamic nature; fluctuations in the application workload inherently come along with variations in scheduling information. An accumulation facility striving to provide an adequately detailed record on these characteristics will face high frequencies of scheduler-relevant events, and a resulting abundance of scheduling data. Consider for example a multi-threaded web server, with a main thread, and several "worker threads", each devoted to a particular client connection. Tracking communication patterns or processing time of all these execution contexts requires the instrumentation of frequently invoked operations such as inter-thread communication or thread context switches. Improper or inefficient accumulation will therefore have serious implications on performance, both through the additional execution overhead, and through the high volumes of generated scheduling data.

To allow accumulation of scheduling data even at performance-critical events, and to control the amount of data that flows through our logging infrastructure, we have incorporated several data aggregation and reduction techniques within our logging infrastructure. While the main purpose of these techniques is to improve efficiency, they may improve flexibility as well, since they can be used to answer specific scheduling questions. For instance, a scheduling policy may wish to find out the job that has run on a given processor most recently. However, it may require neither exact timing characteristics of this job, nor a complete histogram of accesses to the processor by *all* jobs. Rather than being forced to sort out unwanted data itself, a scheduler can use our mechanisms to aggregate or filter data at the source.

We based the design and selection of appropriate aggregation mechanisms mainly on the criterion of efficiency. All in-place aggregation should be simple and cheap; implementation of complex and costly mechanisms is left over to the scheduler application. The incorporated mechanisms mainly focus on *clustering* data, that is, grouping of log data sets by matching values, and on *omitting* unnecessary or unwanted data. We refrained from integrating any statistical methods within the logging infrastructure; such methods are usually more complex than the previous two techniques. Also, statistical methods typically operate on sets of data, whereas clustering and omitting data operate on a per-sample or per-event base. Consequently, while the latter techniques clearly have advantages if being done before or during accumulation, we see no compelling reason to perform statistical calculations instantly, since they can only be applied *after* the accumulation of a complete set has taken place anyway.

3.6.1 Clustering Data

A simple technique to reduce log data is to group entries by matching values. Since the matching entry is then dispensable, it does not need to be stored with each log entry. We refer to this technique as *log clustering*. The restriction to only five different types of log entries limits the degree of freedom for clustering log data: data can be grouped by (i) the event identifier (i.e., characteristic), (ii) the principal's domain, (iii) the counter, (iv) the correspondent's domain, and (v) the time stamp.

To implement per-event and per-domain clustering, we extend our support for *separate* log files to scheduling characteristics *and* accounting domains. In other words, we provide a mechanism to route data streams from distinct characteristics and domains to dedicated log files. Concretely, our facility maps each pair *<event identifier, accounting domain>* to a separate log file selector, which in turn refers to a designated log

file. For global characteristics such as system load indicators, the accounting domain is omitted and the log buffer is resolved solely based on the event identifier. When accumulating a given scheduling characteristic, the log mechanism resolves the mapping and dereferences the log file selector. It finally writes the respective information into the resulting log file.

As with the associations between domains and principals, the task of establishing mappings from event identifiers and domains to log files is exported to the scheduler application. Using the control interface, the scheduler can further define the particular contents of each log buffer, and thereby avoid the dispensable matching event and domain entries. Since the specification of the log layout is done on a per-log file base, contents and sizes of a log file are guaranteed to be fixed, no matter if the log file contains data from a single, or multiple sources[1]. With this scheme, several types of log files and layouts are possible, as for example:

A global log that can be used by all events and domains. A potential application for this type of log file would be a global history of characteristics on a given processor.

A per-event, per-domain log, used to store exactly one scheduling characteristic of exactly one domain in a dedicated log file. Such a log buffer could for instance hold a history of the domains' processor service demands.

A clustered domain log, used to store data for clusters of domains. Clustered domain logs can be beneficial if the scheduler has a priori knowledge of relations between different domains. In this case, the scheduler can employ group-wise log files to keep together scheduling data of closely related domains.

So far, our facility only allows for clustering data per characteristic, and per domain. The linear nature of event-logging implicitly provides support for sorting data chronologically, and no extra support is needed to cluster data by time. Further, the different semantics of the counter type renders clustering data by this entry completely useless. In contrast, aggregating data by correspondent domain is a useful technique in cases where a scheduling policy requires data records on a per-event, per-domain, and per-correspondent base. However, we refrained from incorporating this technique within our framework, since it would require implementing a three-stage routing mechanism, complex both with respect to execution and maintenance. Future work has to be done to explore, if and how efficient data structures and algorithms capable of aggregating correspondent-wise data can be incorporated into our infrastructure.

3.6.2 Omitting Data

While log clustering makes some of the data dispensable, it does not affect the actual information contained in the data. The next class of mechanisms is concerned with omitting log data *and information*, by recording only a subset of the data. Such techniques are useful in cases where the data is too high in volume for being considered completely anyway. The basic idea is to discard some of the relevant information, in the hope that the remaining information will still be sufficient and accurate enough to make proper scheduling decisions. Our logging facility employs four basic techniques to reduce the volume of accumulated scheduling data in-place: domain-based filtering, cyclic logging, event counting, and sampling.

[1] If a given entry has meaning only for a subset of the data sources, the log mechanism uses dummy place holders for the rest.

Filtering Data using Accounting Domains

Accounting domains decouple the resource principal from the accounted entity reported to the scheduler, by allowing for dynamic associations between principals and domains. Conversely, each domain can be considered as a pool of resource principals the scheduler considers to be "equal" with respect to accounting. Thus, the domains implicitly define a filtering instrument, since the logging facility can ignore all events that reflect state changes *internal* to an accounting domain. We implement this filtering instrument by having the log mechanism check if changes in resource consumption take place within or across accounting domains, and letting it disregard the internal changes. As an example, if two threads reside in the same domain, context-switches between these domains are not regarded as resource release and access anymore. Rather, the same domain continues to access the processor, and the log mechanism ignores the event. Likewise, the mechanism ignores all resource sharing within a particular domain, and only records accesses to those resources, which are shared among multiple domains.

Cyclic Logging

Compared to performance analysis and system debugging, where event-logging usually is used, scheduling decisions are time-critical, and done comparatively frequently, in the order of magnitude of milliseconds. The scheduling information basically serves as a means to predict the near future based on the past, and the relevance of scheduling correlates with the period between scheduling decisions; in other words, the log data usually becomes obsolete quickly.

For this reason, all log files in our infrastructure are based on cyclic buffers. The log base pointer rolls over to the first entry when being incremented from the last entry. The oldest entries in the log file will automatically be overwritten with new data when the buffer is full. It is the responsibility of the scheduler to make the buffers adequately large. Likewise, if a scheduler application wishes to preserve log data, – e.g., to store profiles in a database [28] –, it must take care of the preservation itself. If the log files are not large enough to prevent data from being overwritten, the scheduler must periodically copy the log buffers into a user-level database.

Event Counting

Another technique that can be used to reduce the amount of collected data is to *count* the times a particular event has occurred, instead of logging each event separately. While there are scheduling characteristics whose access counters are only available in a cumulative form (typically those, which are accumulated at on-entry/exit or periodical events), counting can also be used as a *specific mechanism* to reduce data flow [2]. A simple realization of an event counter is a cyclic additive log file with only one entry. To implement this scheme, we extended the log control interface to the scheduler by a method that tells the log mechanism to *add* new entries instead of overwriting them. For counting an event, the scheduler can simply route it to a log file that consists of only one entry, and configure the log file to store only the counter value. Toggling the "add bit" will then cause the log mechanism to count the number of occurrences of the event in the specified entry.

Sampling

Finally, we also provide support for sampling of scheduling data. Sampling basically means to "select a subset of the observed data for analysis, rather than all" [37]. Choosing the appropriate subset can be accomplished in multiple ways; one can perform a random selection, or consider only the most frequent, or most meaningful parts. The majority of sampling techniques is applied to generated data. One method that can be implemented in-place is *event-based sampling*, that is, to consider only a subset of the events for logging, instead of all. In contrast to the previously introduced mechanisms to omit data, event-based sampling tries to reduce the frequency of observations, rather than the amount of log data. Our logging infrastructure features an event sampling mechanism based on *thresholds*. The thresholds act as a filter for log events: the number of events identical to the threshold must be reached, before a log entry is made. Like all log controls, the threshold configuration is exported to the scheduler application, allowing it to modify the event thresholds according to its own demands.

Generally speaking, the event threshold must reflect the speed of alteration of a particular scheduling property. The more dynamic a given property, the more frequent is the occurrence of the corresponding event. For global properties such as system load indicator, it is sufficient to employ one threshold per event. However, resource usage is principal-specific, and may also vary among different *principals*. Consider for instance a server application that consists of a set of frequently interacting threads, which runs concurrently with a "number-crunching" application that consists of rather long running threads. In this case, associating thresholds only with the log events results in the log file being polluted with characteristics from the former application, while data of the latter is filtered out and lost for the most part. For resource usage, we therefore employ one threshold per event *and* accounting domain. Although this incurs more overhead for the scheduler in order to program the logging facility, maintaining per-event and per-domain thresholds can be of great benefit for monitoring resource usage in pace of the principal-specific fluctuations.

The log mechanism itself does not modify the thresholds set by the scheduler. If the scheduler uses fixed threshold values over time, the subset selected for logging follows the constant frequency resulting from the threshold. Sometimes however it is advantageous to alter the frequency randomly, in order to avoid pathological cases and to improve the representativeness of the data. Since the scheduler can modify the thresholds, it can easily conduct random sampling by feeding the thresholds with values varying randomly over time.

3.7 Log Data Access and Scalability Considerations

The logging facility collects all scheduling characteristics within designated, per-component log buffers, held in main memory. All collected characteristics are eventually promoted to the scheduling application, which analyzes the data to base its allocation decisions on. Several problems arise related to the writing, and reading of log data and meta-data. First, concurrent accesses to log data introduce potential synchronization and integrity problems, and require us to provide appropriate coordination among the accesses. Second, as the producers and consumers may reside on different processors, scalability problems will arise, if our mechanisms and data structures are not designed appropriately. However,we must not trade off efficiency when addressing synchronization and scalability issues; otherwise our approach will fail to meet the demands of

frequently and timely performed scheduling decisions. For this reason, we do not want to require log data producers to hold any costly interprocessor synchronization operations while making log entries. Finally, our infrastructure must address the question, who should be held responsible for providing the memory regions backing the log data. In the following section, we will discuss the issues related to log data access and sharing in more detail, and show how our logging infrastructure addresses these issues. We begin by stating a set of general requirements on scalability the logging infrastructure must adhere to. We then explain how the log architecture achieves the scalability requirements and addresses the arising synchronization problems, both with respect to log data accumulation and sharing. We finally discuss the question of adequate provision of memory to back the log data.

3.7.1 Scalability Requirements

In [64], Unrau et al. describes three general guidelines for the design of a scalable operating system: (i) preserving application parallelism, (ii) bounded overhead with respect to the number of processors, and (iii) preserving locality of applications. Although accumulation of scheduling is not bounded to the operating system kernel, but can also be applied to application programs, we nevertheless consider it to be a part of the operating system and thus subject to these guidelines. We therefore translate the three general principles into three concrete requirements for our approach: first, we require that there is a separate data accumulation process on each processor. (Since we do not want to restrict the design of the scheduler extensions, we do not explicitly require separate *analysis processes* on each processor – although research has shown that a decentralized scheduler design turns out to be more efficient in general [24, 30]). Second, we require that accumulation and analysis of log data of one processor does not affect other processors than the involved ones. Third, we require that all log buffers are held in local, per-processor memory.

3.7.2 Data Accumulation

Since logging code constructs do not contain parallelism in their semantics, we can easily implement independent, per-processor accumulation processes that avoid interprocessor control transfers. Further, we deploy independent, *per-processor log buffers*, which are kept in processor- or node-local memory. Remote memory is only accessed when a analyzing log data, but not when producing log data. The idea of processor-wise log buffers is implemented by many tracing and debugging facilities [2, 16, 66]. Keeping log data and meta-data in processor-wise and local memory not only prevents costly latencies of remote memory references and removes the overhead of cache coherency protocols; it eliminates the requirement for synchronization operations between log data producers on different processors. Since we additionally separate log data in *component-wise* log buffers, we must only address the problem of multiple log data producers that reside within the same component *and* on the same processor. However, since the problem is local to each processor, we can easily keep the producers coordinated without the need of costly inter-processor synchronization operations. A simple solution is to guarantee that log operations are atomic. Within the kernel, one can easily achieve this by disabling preemption. At user-level, one can ensure atomicity by means of a preemption avoidance protocol [38, 63]. Alternatively, in cases where disabling or avoiding preemption is not possible, one can resort to atomically *reserving* log entries instead, for instance by using a single instruction to increment the index.

3.7.3 Data Sharing

To supply the schedulers with the accumulated properties, the log infrastructure must provide them with access to the log files. Tracing facilities often copy-out the data, or use double-buffering for this purpose [2,16,68,69]. Copying and buffering are effective mechanisms to ensure correctness and completeness of the gathered data, since they relieve the infrastructure from having to deal with synchronization or data corruption problems. However, such techniques incur heavy performance costs, not only with respect to the execution overhead, but also in terms of pollution of processor caches and translation look-aside buffers. Especially for the time-critical task of scheduling, where good performance is a firm requirement, we do not consider them as tolerable solutions. We instead *share* the log buffers *directly* with the scheduler subsystem, and memory-map all log buffers into the scheduler's address-space. Since the size of scheduling data is rather small compared to performance traces, direct sharing is a practical and efficient approach, reducing both cache footprint and space requirements of logging.

However, sharing log buffers implies the possibility, that log data is written into a log file being analyzed simultaneously. Care must be taken to coordinate concurrent reads and writes to log memory. However, for performance and scalability reasons, we do not want to require analysis and accumulation processes to synchronize each other before performing their respective accesses. Instead, we let the data producers and consumers run unsynchronized, but enable the analysis process to *detect* reading of inconsistent data. More concretely, loss of consistency occurs, if the log analysis process reads an entry that is currently being written by a logging operation. Thus, as described in [67], one can detect the potential inconsistency by associating a *valid bit* with each log entry row, which is set by the log data producers after entering entries into that row, and cleared by the analysis process after reading the row.

In our facility, we use a modification of this scheme, which slows down the more seldom case of concurrent analysis and log producers, but eliminates the requirement for valid bits in the frequently executed log operation: instead of using valid bits per log entry row, we use the log file's *current index* to detect if a log producer has made entries to that file while the analysis process was reading it. That is, before parsing a log file, the analysis process saves the log file's current index in a local copy. Having finished parsing the file, the process verifies that the current index has not changed meanwhile, by checking it against the local copy. If they do not match, it must restart the *complete* analysis of that file, since it may have read inconsistent data during analysis. Since log buffers are processor-local in our facility, the log producers do not need to use memory barriers or other synchronization operations when incrementing the index; the only requirement is a write ordered memory consistency model, in order to ensure that the the index always reflects the state of the log file consistently. Additionally, this scheme requires to detect the corner case in which the producer has written exactly as much entries as there is room in the complete log file. Further, with increasing probability of collisions between reader and writer, the scheme requires a means to prevent the analysis from starving forever. See Section 4.5 for all implementational details on this scheme.

3.7.4 Memory Provision

The last problem discussed in this section is the question, who should be held responsible for providing the memory used to store log data. Generally speaking, it should

be the duty of the scheduling subsystem to page the instrumented components with log memory. Since the scheduler subsystem is an application that resides at user-level, such a scheme principally requires kernel support for application-controlled memory management [6, 34]. Moreover, if the instrumented component is the operating system kernel itself, it also requires a solution, to hand over kernel-memory management to the user-level [31]. In cases where such mechanisms are not supported, we propose to allocate pinned memory for log buffer regions, in order to avoid the trust problem that arises if the scheduler tries to read a log file that is not mapped into memory.

Chapter 4

Implementation

4.1 Overview

To evaluate accumulation of workload characteristics via logging, we have implemented a prototype logging facility within a real-world, microkernel-based component operating system. Like depicted in figure 4.1, our environment consists of four major parts: (i) a small, privileged microkernel; (ii) several operating system modules running at user-level; (iii) the application workload; (iv) a scheduler application responsible for processor allocation to applications and driver modules. The software stack runs on a medium-scale (up to 16-way) IA-32 architecture based multiprocessor.

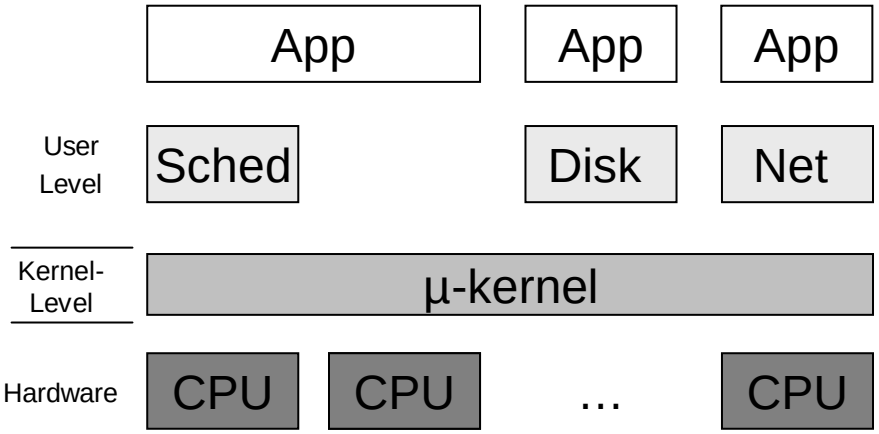

Figure 4.1: Component operating system environment

Our logging facility extracts different workload characteristics from the microkernel, the operating system components running at user-level, and the application workload. We have implemented a prototypical scheduler application responsible for processor allocation to applications and device drivers. The scheduler can analyze the characteristics recorded in the shared log files to perform its actual allocation and migration decisions. However, since our work focusses on supporting scheduling, not on investigating different scheduling strategies, we leave the actual scheduling policy rather open, and focus mainly on the scheduler's task of evaluating the log files.

The following sections describe our prototypical implementation and its details. We begin with outlining the components our environment consists of (Section 4.2). Subsequent sections present the implementation of the logging facility. We first show, how we instrumented the privileged microkernel (Section 4.3). We then present the instrumentation of the application workload and user-level operating system modules (Section 4.4), emphasizing however on the differences and particularities with respect to the microkernel instrumentation. We finally present the implementation of the user-level scheduler application (Section 4.5).

4.2 Environment

Our environment is a microkernel based operating system, which provides its services to the application workload. As microkernel, we use an implementation of the *L4 microkernel* developed at the University of Karlsruhe. To host the application workload, we leverage *L4Ka virtualization technology*, also developed at the University of Karlsruhe. The L4Ka virtualization architecture basically consists of paravirtualized Linux instances – dubbed *L4Linux* – , which run in unprivileged user-mode, on top of L4. We make use of virtual machines not only for hosting the application workload; we use this technology also for providing user-level device driver services to the applications. The following paragraph will shortly describe the core functionality and abstractions of the L4 microkernel, and of the L4Ka virtualization environment.

4.2.1 The L4 microkernel

L4 is a state-of-the art, second-generation microkernel, which provides a minimal set of powerful abstractions that can be used to build extensible systems on top [44]. L4 has three core abstractions: threads, address-spaces, and inter-process communication (IPC). Threads are the basic notion of a schedulable entity exported by the L4 microkernel. Also, they serve as the endpoint of the inter-process communication provided by L4. Address spaces are first class kernel objects in that the hardware aspects are implemented within the kernel. However, management of virtual address spaces can be done outside the kernel, at user-level [43]. The IPC mechanism offered by the kernel provides synchronous, unbuffered operations to send messages between threads. As the generic mechanism for interaction between system components, the IPC path is invoked very frequently. Its performance and efficiency is therefore crucial to every L4-based system [42].

L4 features an internal scheduler that dispatches threads using a priority-based round-robin scheme. Each thread has a time-slice that determines the length a thread can run on a processor before being preempted by another thread. L4 provides an interface that allows the control over scheduling from user-level. Using this interface, a user-level scheduler can modify scheduling parameters such as thread priorities, time slice lengths, or the particular processor a given thread should run on. Changing the last parameter will cause L4 to migrate the thread to a different processor instantly.

4.2.2 The L4Ka Virtualization Environment

The L4Ka virtualization project investigates how microkernel technology can be applied to virtual machines [60]. The virtualization architecture consists of a framework

that allows paravirtualized guest operating systems to run on top of L4. The guest operating systems are basically adoptions of the Linux kernel, modified to run on top of L4, instead of on bare hardware [33]. The framework additionally provides a virtual machine monitor that implements the virtualization layer the L4Linux instances. The monitor is an L4 task running at user-level. In addition to the guest virtualization, the monitor also acts as a resource manager, responsible for allocation of memory, processing time, or device I/O to the virtual machines. The monitor uses L4 abstractions and mechanisms for virtualization and resource allocation.

The virtualization framework basically allows us to run any Linux application within our scenario, unmodified, but encapsulated within a virtual machine. Beyond hosting the application workload, the framework also provides the device driver modules running at user-level. The driver modules are essentially L4Linux instances that encapsulate and reuse Linux device driver logic [41]. Every driver exports a virtual device interface that can be used by the application workload to request driver services. The driver then multiplexes these requests on the physical devices it manages. Interaction between applications and drivers are based on the user-level memory management primitives and the IPC mechanisms provided by L4.

4.3 Instrumentation of the microkernel

In this section, we demonstrate the applicability of our logging concepts and mechanisms, by showing, how these concepts were implemented within the L4 microkernel. The microkernel is the entity in the system that runs in privileged mode; further, some of its services are invoked very frequently. We must therefore take special care when instrumenting the microkernel, in order to avoid safety and efficiency problems affecting the overall system's performance and stability.

The following paragraphs describe the microkernel instrumentation in more detail. We begin with describing scheduler-relevant characteristics that accrue within L4 (Section 4.3.1). We then detail the dynamic instrumentation techniques we use to safely enhance the kernel with log handlers (Section 4.3.2). Subsequent sections present the implementation of log handlers (Section 4.3.3), the accounting domain concept (4.3.4), and the data aggregation techniques 4.3.5). We finally discuss implementational issues of log data access (Section 4.3.6).

4.3.1 Workload Characteristics Accruing in L4

L4 only offers a minimal set of core abstractions, and tolerates a concept inside the kernel only if moving it outside prevents the system from working correctly [43]. The minimality of the L4 microkernel limits the workload characteristics internal to the kernel. Also, L4 offers only basic services that can be used to build more elaborate system functionality on top. Consequently, some of workload characteristics accruing within the kernel can also be extracted from user-level components implementing higher-level services or functionality. Following the microkernel philosophy, workload characteristics should be extracted only from within the kernel if not possible otherwise.

We have identified five different characteristics relevant for scheduling that can be measured within L4: (i) processor service time, (ii) information from performance counter hardware, (iii) IPC usage, (iv) memory usage, and (v) changes in the runqueue length. In the following paragraphs we will discuss each of these characteristics

in more detail. For each characteristic, we detail its relevance with respect to multiprocessor scheduling. We further specify appropriate log events and describe how the characteristics can be measured at these events. We also discuss the question, if the characteristics are kernel-specific, or if they could also be measured at user-level.

Processor service time. As the central resource with respect to scheduling, knowledge of the processor service time is of interest for many scheduling policies [9, 15, 28, 40]. There exist different attributes and measures related to processor service time, such as a principal's aggregated time demands, or the most recent time a principal has been serviced by a processor. Also, idle times are relevant within the context of scheduling.

The basic notion of a processor principal in L4 is a thread. To measure service time, we therefore instrument the context-switches between threads (see Figure 4.2). A context-switch is basically the *exit event* for the preempted thread, and the *entry event* for the dispatched thread. For the exiting thread, the kernel calculates the time spent running on the processor based on its entry and the current time. It emits the preempted thread's service time to the designated log file. For the entering thread, the kernel records the current time in an internal buffer variable. Depending on the desired accurateness, the kernel bases time calculations on the IA-32 time stamp counter register, or on the internal timer tick counter[1].

Thread switches are on the critical IPC path in L4; measuring timing properties can therefore significantly reduce the performance of the microkernel. In Section 4.3.5, we discuss how we implemented the aggregation techniques that strive to reduce this overhead. Unlike other workload data, timing information only accrues within the kernel. This stems from the fact, that kernel thread switches are safety-critical operations that must be performed by the kernel – at least if the threads reside in different address spaces [45]. Also, switches are not necessarily the direct result of a user invocation, which could be instrumented at user-level instead of the switch in the kernel.

Performance counter information. The Performance Counters of the IA-32 architecture provide a rich source of information on the current state of the processor [36]. The most palpable examples of using performance counters for scheduling purposes are cache affinity policies that monitor the number of cache-misses [12, 58, 65]. But performance counters also provide other information such as mean cycles-per-instruction [1, 29].

Scheduling policies typically require performance counter information on a per-principal base. In L4, these principals are, as with processing time, the kernel threads. Since performance counters are maintained by hardware, and oblivious to the kernel's notion of threads, we must multiplex the counters among the kernel threads. More concretely this means that, again, the context-switches must be used as instrumentation points. The kernel calculates a counter "delta" for the preempted thread, and buffers the current value for the freshly dispatched thread. Alternatively to context-switches, the performance counters could be monitored at asynchronous events such as the periodic timer tick. At these events, the kernel

[1] Currently, this can be specified at compile-time. To specify the timing base at run-time, we propose to declare two separate events for the different timing bases. The scheduler can then select the desired timing using the different event types at run-time.

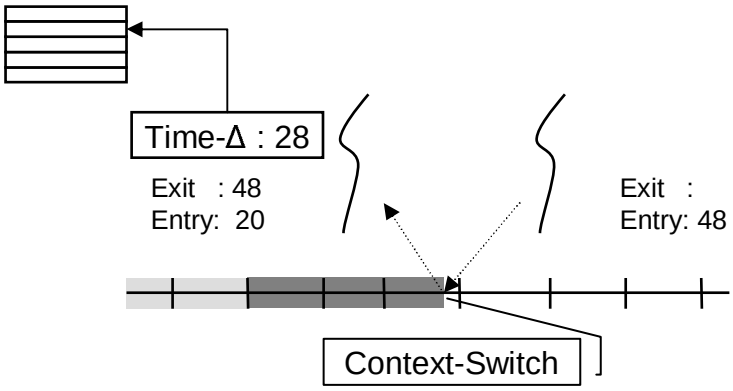

Figure 4.2: Measuring timing properties on context-switches. The log facility reports the service time of the preempted thread to a log file and buffers the entry time of the freshly dispatched thread.

would have to determine the counter and the currently running thread, and record the information within the corresponding log buffer. At present, we instrument the context-switches, since only this approach provides an accurate record of per-thread performance counters. The scheduler can still do coarser-grain logging, by using the offered aggregation and sampling mechanisms.

IA-32 processors can be configured to permit user-level accesses to performance counters. Thus, at a rough glance, there should be no requirement at all to monitor the counters within L4. However, performance counter values are only useful data if monitored per principal. The scheduler depends therefore on the kernel if it wishes to retrieve per-(kernel-)thread performance counter information.

IPC usage. L4's IPC facility is a shared kernel resource. IPC usage patterns can thus be used to identify relations between principals, which is an important information for of co- and gang-scheduling strategies [25,51,56]. However, IPC usage is also useful to identify affinity to processor-associated resources. For example, a driver managing a device that is associated to a processor is also dedicated to this processor; a high IPC interaction between an application and the driver implies a high affinity of the application to the processor.

L4 provides a synchronous unbuffered IPC mechanism between one sender and one receiver thread. We instrument code path to track invocations of the mechanism. To ensure correct logging records, sender and receiver must log usage of IPC separately. This stems from the fact, that sender and receiver may reside on different processors; in this case, the sender cannot access the receiver's log buffers, and vice-versa. We therefore instrument the IPC implementation twice – the first time on the sender-side, as soon as the sender has transferred the message and the operation is known to succeed, and the second time on the receiver side, as soon as the receiver has been reactivated by the sender due to the successful transfer of the message.

The IPC mechanism provided by L4 is a very minimal, but highly efficient communication operation. The idea behind that is, that other, more complex communications mechanism can be built on top of the IPC mechanisms. Consequently, communication patterns may also be monitored by instrumenting high-

level communication abstractions at user-level. If user-level instrumentation is not possible or too complex, a scheduler must fall back to kernel IPC statistics.

Memory usage. In the context of scheduling, memory usage statistics can be used for two types of scheduling policies. The first type of policies requires information on *allocations of memory* [10, 52]; the second type bases its policies on *usage of shared memory* [11, 56].

L4 exports management of virtual memory to the user-level. It therefore provides operations to *map* and *revoke* virtual memory between different address-spaces [43]. To track these allocations, we maintain memory page counters monitoring the number of pages allocated to address spaces (i.e., the principals of virtual memory). The counters are incremented and decremented whenever a memory page is added to or revoked from an address space. Different events are appropriate for logging allocation counters: the *access events*, that is, the map and revoke operations, or other, *asynchronous events*, such as the periodic timer tick. At present, the only kernel offers to log the counter at the access event. That is, after the successful mapping or revoking of a memory region, the counters are updated and then emitted to the log files.

To ensure correctness of mappings and revokings of memory, the kernel maintains a mapping database. This is a tree-like data structure that records all map operations, to ensure that the mappings can be removed correctly afterwards. The mapping database can also provide information on which pages are shared between multiple address spaces. Page references are tracked by the IA-32 hardware, in kernel-internal page tables. Thus, all required information to monitor shared page usage is available within L4. However, three problems prevented us from instrumenting the kernel in order to monitor usage of shared pages: first, the layout of page tables is predefined, and parsing these tables is principally a complex operation, which requires a significant amount of time. Second, since page reference bits are tracked by hardware, there is no knowledge on the pace of page usage that would be available to the kernel. Third, the kernel cannot differentiate between actively and passively shared pages, as the actual management of virtual memory is performed at user-level. However, for scheduling purposes, passively shared pages are of little interest, as they are used for instance to share code, but not for communication. Monitoring shared page usage within L4 would therefore not only require the inevitable parsing of page tables; the parsing may often result in no useful information at all, since the pages may not have been modified since the last scan, or are shared passively anyway.

We therefore chose to monitor shared memory at user-level only. In contrast to some of the previous workload characteristics, knowledge on memory usage is completely available to the user-level. User-level memory managers principally possess some knowledge on memory sharing [6], and L4 provides an interface to query reference bits (i.e., usage) of memory pages. In Section 4.4, we demonstrate an implementation at user-level, by tracking shared memory usage in the device driver components of our environment.

Run-queue length. If an operating system maintains per-processor run-queues, the lengths of these are a commonly used indicator of the current load distribution [8, 53]. Although a simple indicator, research has shown that this relatively simple measure yields even better results than other, more complex ones [39].

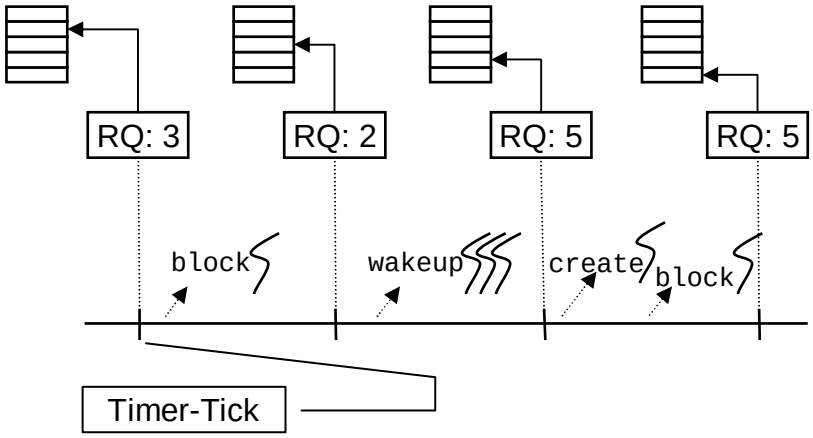

Figure 4.3: Measuring the run-queue length at timer interrupts. We maintain a per-processor variable that reflects the current state of the run-queue. On each timer interrupt, the log handler emits the current run-queue length to the respective log file.

For its own thread scheduling, L4 maintains kernel-internal, per-processor run-queues. However, for efficiency reasons, the kernel uses a *lazy* scheme to manage the run-queue. That is, L4 avoids costly enqueueing and dequeuing operations on the critical IPC path, and uses per-thread flags instead to mark the correct scheduling state. The kernel clears the pending backlog of postponed changes whenever it parses the run-queue to search for a ready to run thread [42]. To reflect these silent changes in our logging facility, we maintain a per-processor counter representing the *real* length of the run-queue. This counter is updated on all state changes in the kernel that imply a increase or decrease in the run-queue length. We thereby ensure that the run-queue length is always reported correctly, still avoiding however the costly modifications to the run-queue on the fast path.

We have identified two different event types appropriate for logging the run-queue length. First, the length can be logged the instant it changes. Second, the length can be logged periodically, every timer tick. Since the IPC invocations potentially change the run-queue length, we refrained from logging the run-queue length on every change. Like illustrated by Figure 4.3, we rather chose to instrument the timer interrupt to reporting report the length of the run-queues. Since re-balancing decisions happen comparatively infrequently, in the range of hundreds of milliseconds, the timer interrupt, firing every millisecond, still occurs often enough to make it possible to perform proper re-balancing decisions.

Principally, the number of ready to run kernel threads is only available within L4. The microkernel is responsible for thread scheduling, and performs this task obliviously to the user-level. Load estimation based on the number of runnable L4 threads therefore relies on extracting this information from the kernel. However, there may exist other notions of threads in the run-time system, for instance if the application workload uses user-level thread libraries. The scheduling state of these threads does not necessarily coincide with the state of the L4 kernel threads. Depending on the workload, load estimation must therefore be based on kernel-, or on user-level threads, or on both. As will be described in Section 4.4,

the scheduler application in our environment can retrieve both the length of the L4-internal and the virtual-machine-internal run-queues.

4.3.2 Dynamic Instrumentation of Log Events

The following Section details how we instrumented the L4 kernel to efficiently transfer control to the log handlers. Efficiency of the microkernel is one of the key requirements in a microkernel based system [33]. It was our goal was to keep the instrumentation overhead low as possible.

Many of the discussed log events on the critical IPC code path. Besides the obvious case of the IPC event itself, measuring timing characteristics or performance counters is also done during IPC, since the synchronous nature of the IPC mechanisms implies a context-switch between sender and receiver thread. Dynamic instrumentation avoids burdening the hot spot of the microkernel with disabled-logging checks, and our facility makes use of this technique to allow the scheduler to choose the characteristics at run-time. If logging of a characteristic is enabled, the kernel determines all corresponding instrumentation points its code image. It then modifies these points and injects a control transfer to the logging logic, executed whenever the instruction stream reaches a particular instrumentation point.

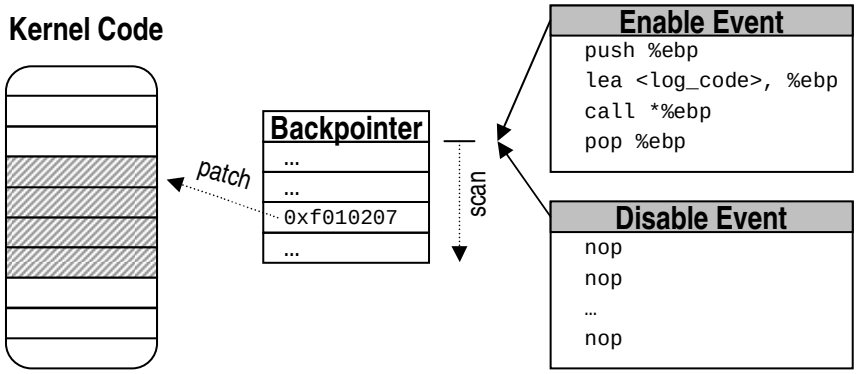

Figure 4.4: Dynamic instrumentation of log events. The instrumentation points are prepared with a small sequence of `no-op` placeholders; further, the points are stored in a per-characteristic backpointer list. When enabling logging of a particular characteristic, the kernel parses the backpointer list, and injects the respective instructions to transfer control to the log handlers.

In contrast to other dynamic instrumentation tools [16, 35, 59], our solution does not need to instrument *arbitrary* machine instructions. The possible instrumentation points in our system are limited to the log events, which are known at compile-time. We are therefore able to *prepare* these points for later dynamic instrumentation. We use a method similar to the `alternative_input()` function in the Linux 2.6 kernel. Our method is depicted by Figures 4.4 and 4.5, and will be described in the following. At each of the instrumentation points, we enhance the source code by a small assembler macro (Figure 4.5) that performs two tasks: Firstly, it inserts sequence of `no-op` instructions, to make room for later control transfer to the logging code. If the invoked logging code requires arguments to be passed, the macro inserts additional `no-op` place holders, later substituted with instructions passing the arguments via the stack. Second,

the macro tells the linker to save the start and end address of the `no-op` sequence in a back-pointer list. This back-pointer list stores the instrumentation addresses on a per-characteristic base. The instant the scheduler enables logging of a characteristic, the kernel parses the list of back-pointers to the `no-op` sequences. At each of these addresses, it patches the `call` instruction transferring the control to the logging code, and additionally inserts the `push` and `pop` instructions that pass the required arguments via the stack. To avoid additional instructions in the original code path, we do not respect any compiler conventions on register savings when transferring control to the logging code; instead, the called code preserves all registers itself, including the caller-saved ones.

```
#define DECLARE_LOG_EVT(evt_id, log_function, arg1)                    \
__asm__ __volatile__(                                                  \
    "1:                                  \n\t"                         \
    ".byte 0x8d,0xb6,0x00,0x00,0x00,0x00 \n\t" /* 6-byte no-op     */ \
    ".byte 0x8d,0x74,0x26,0x00           \n\t" /* 4-byte no-op     */ \
    "2:                                  \n\t"                         \
    ".section .log.evtenable."#evt_type" \n\t"                         \
    "push %0                             \n\t" /* push arg         */ \
    "lea "#log_function", %%ecx          \n\t" /* load call addr   */ \
    "call  *%%ecx                        \n\t" /* transfer control */ \
    "pop %0                              \n\t" /* pop arg  */         \
    ".previous                           \n\t"                         \
    ".section .log.evtlist."#evt_type"   \n\t" /* backptr list     */ \
    ".align 16                           \n\t"                         \
    ".long "#evt_type"                   \n\t" /* store type       */ \
    ".long 1b                            \n\t" /* store begin      */ \
    ".long 2b                            \n\t" /* store end        */ \
    ".previous                           \n\t"                         \
    :: "r" (arg1) : "ecx");
```

Figure 4.5: Assembler macro that prepares the dynamic instrumentation. The macro inserts the required `no-op` place holders into the original code, and then stores the control transfer instructions within a different section. Finally, it stores the begin and end of the place holder sequence in the backpointer list.

By default, logging is disabled, and the code only executes the few inserted `no-op` instructions before it continues normal execution. The costs for disabled events is thus limited to (i) the cycles of additionally executed `no-op` operations, (ii) the space these `no-ops` occupy in the instruction cache, and (iii) the potential costs for preparing input arguments for being passed via the stack. Only a few `no-op` instructions are required, which renders the former two costs minimal and tolerable even on the critical IPC path. The costs for preparing arguments arise only if the arguments are not held in processor registers at the instrumentation point. In this case, the arguments must be fetched into a register before being saved on the stack by the `push` instruction. The register convention is determined by the compiler, thus the fetching cannot be instrumented dynamically. It may involve memory accesses and also waste register real estate. However, in the common case, this step should not be required. The arguments passed to the logging code specify the workload data to be logged at this event. If the event is synchronous with the accrual of the workload data (which holds definitely true for all on-access/on-change, and on-entry/on-exit events), the arguments are very likely to be used by the original control flow anyway. On the critical IPC path, we only

passed parameters that were already held in registers[2].

The kernel provides a system call interface that can be used by the scheduler to switch on and off logging at run-time. On invocation of the system call, the kernel changes all corresponding code addresses according to the scheme described above. At present, all multiprocessor instances of L4 share a single code image in memory (For versions with per-processor images, one could establish temporary mappings to modify code sections residing in remote memory). In any case, there is certain chance that the code image is modified while simultaneously being executed on a remote processor. To ensure that the dynamic instrumentation is safe, we must avoid execution of inconsistent code. We therefore atomically patch in a `jmp` instruction at the beginning of the code fragment before inserting or removing the actual code. The `jmp` ensures that other processors jump over the potentially inconsistent code fragment while being modified. The `jmp` is removed as the final step.

4.3.3 The Log Handler

This section details the implementation of the actual log handler in the L4 microkernel. Its functionality can be split into the following four sub-steps:

1. Checking thresholds
2. Retrieving and parsing the log control register
3. Accumulating the log data
4. Logging the data into and updating the control register

The invoked log code checks at first, if the log thresholds have been reached and logging should take place at all. The threshold functionality will be described together with the other data aggregation mechanisms, in Section 4.3.5. In the next step, the log mechanism retrieves the *log control register*, a data structure that is associated with each log file. The mechanism afterwards accumulates the data to be logged. It finally writes the data out into the designated log buffer, and performs the required updates to the log meta-data in the log control register.

Log control register

While the log events themselves are predefined at compile time, the log files and their contents can be specified at run-time. We implement this scheme by representing each log file through a *log control register*, which contains all meta-data associated with the log file. The register denotes the memory buffer reserved for logging, and the location of the next log entry within the buffer; it also specifies the entries to be stored in the log file. We co-locate the control registers with the actual log buffers, in the shared log area. To implement the log control interface, we permit the schedulers write-accesses to the log area, allowing them thereby to modify the log control registers directly (For all details on log data sharing, we refer to Section 4.3.6). The mappings from events and domains to log files are implemented by means of log control selectors that each point to a distinct log control register. For the run-queue length, we omit the

[2]Since the IPC path is written in assembler, we were also not restricted by the compiler's calling conventions. We could therefore aoivd the stack involvement completely, by passing the arguments *directly* in their registers.

domain indirection and associate the events directly with a control register selector. On invocation, the log mechanism determines the selector corresponding to the given event and domain; it then dereferences the selector to retrieve the log control register. Finally, it parses the register and performs its actual log operation based upon the configuration in the register.

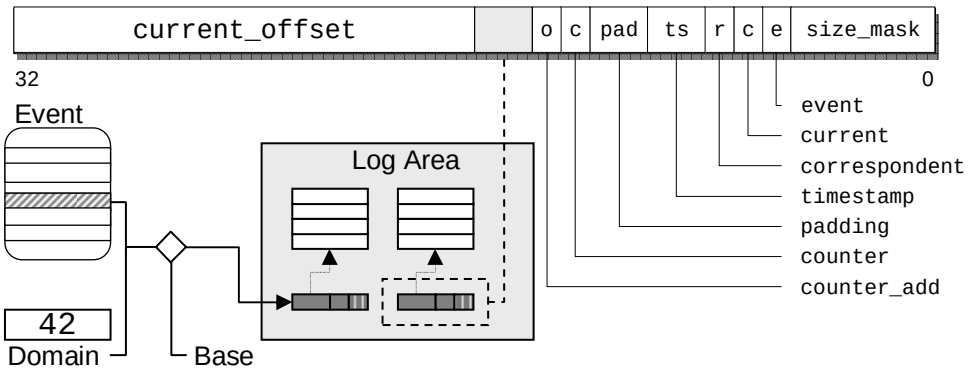

Figure 4.6: Log control registers; each log control register contains the meta-data of a log file such as the current log index and the flags specifying the contents of the log file.

The log control register is depicted by Figure 4.6. It is a 32-bit data structure that contains three members: (i) the current log index, (ii) the size of the log buffer, and (iii) a set of flags that specify the data to be logged in the buffer. The current index points to the buffer that contains the actual log entries. The current index is encoded by a relative offset, `current_offset`, pointing into the shared log space. The current implementation uses 16-bit wide offsets, which limits the log space area to at most 64 KBytes (this is, of course, no design limit). The size of the log file is specified by `size_mask`. For space and efficiency reasons, `size_mask` stores the dual logarithm of the actual size, and the real size is $2^{\texttt{size\_mask}}$. The base pointer of the buffer is not explicitly stored in the register, since it can be calculated by masking the current offset with the bitwise negate of the log file size. The flags implement the content configuration interface. Each of the flags refers to a possible entry, and the scheduler can specify the desired log entries by setting or clearing the flags. Concretely, the scheduler sets or clears the following flags:

Flag	*Description*
`event`	Toggles logging of the event identifier.
`current`	Toggles logging of the domain identifier.
`correspondent`	Toggles logging of the correspondent identifier.
`counter`	Toggles logging of the event counter. The `counter_add` flag is used to implement event counting.

`timestamp`	Specifies the timestamp to be logged. If not turned off, the time stamp entry can be based on three different sources: first, a virtual time stamp can be logged. The virtual time stamp is incremented each time it is read by a log handler. Second, L4's (processor-specific) timer tick variable can be selected. Third, and last, timing can be based on the hardware time stamp counter register. The hardware register is actually 64-bit wide; however, at present, all log entries size are fixed to 32-bit. The mechanism therefore right-shifts the counter by a predefined value (currently set to 8), and only records the lower 32-bit of the result[2].
`padding`	Used to pad a log entry row. To speed up the log operation, the size of a row must be a power of two. The padding specifies the number of dummy entries to be added to a log row. Since at most five true entries are possible, two bits are sufficient for the padding field.

Accumulating the Log Data

Having retrieved the log control register, the log mechanisms scans the control flags, and accumulates the desired data based on the configuration of these flags. This step depends on the particular scheduling information that is to be logged. Measuring hardware properties involves reading the model-specific or other hardware registers. Other characteristics are represented by local variables at the instrumentation point and must be passed to the log mechanism as input arguments. Since passing arguments involves references to stack memory, and potentially wastes register real estate, we try to keep the log mechanism as state-less as possible.

The `event` variable recording the respective identification number is implictly known in the log handler. Time stamps are stored in global variables or hardware registers, and can be obtained without further parameters as well. Accumulating `current`, `correspondent` and `counter` entries depends on the particular scheduling characteristic. We therefore discuss these two entry types separately, for each of the different characteristics in L4.

Processor service time. The service time is logged on thread preemption. The `current` variable holds the accounting domain identifier, which is directly derived from the particular thread. The thread identifier is passed to the log code as an argument. Accumulating the service time stored in the `counter` requires knowledge of the current time and the buffered entry time of the respective thread. The current time is a global variable or hardware register. Since the processor is an exclusive resource, also the entry time can be held in a global variable.

Performance counters. Also the performance counters are logged on thread preemption. The `current` domain is again derived from the preempted thread, which is passed to the log code as an argument. Like with processor time, a per-thread

[2] In other words, the hardware-based time stamp counts the time in steps of 256 clock cycles, and wraps around after 2^{40} cycles. On a 3 GHz Pentium, this yields a times-tamp with an accurateness of about 85 nanoseconds, wrapping around after about 6 minutes. At present, the logging infrastructure ignores inconsistencies that may arise due to the time stamp overflow.

delta is logged as `counter` variable, which requires knowledge of the current performance counter value, and the buffered value of the preempted thread. The current and buffered values are obtained via inspecting the respective performance counter hardware register, using the `rdpmc` instruction. Logging multiple performance counters requires to employ distinct instrumentation points for different performance counters. Since control to the log mechanism is then transferred separately for each of the performance counters, the particular register is known implicitly in the instruction stream.

The semantics of performance counter registers heavily depend on the particular sub-architecture (e.g., Pentium 3, Pentium 4, Opteron). The current prototype provides support for only one event (i.e., one hardware register), and only one sub-architecture (Opteron). Adding more events and sub-architectures should be a simple matter of engineering. To select the hardware events routed into the particular counter, we allow the scheduler modify the performance counter configuration registers *itself*, at user-level. To permit the scheduler to modify these registers, we trap the modifications in the kernel and emulate them appropriately.

IPC Usage. Logging IPC usage is done on access, in the IPC code path. Usage is logged twice, both on the sender and the receiver side. The `current` domain corresponds to the current thread, that is, to the sender on the sender side, and to the receiver on the receiver side. The domain of the communication partner is recorded in the `correspondent` entry. The correspondent domain is derived from the thread identifier of the partner, which is locally known in the IPC code path. The partner's identifier is passed to the log mechanism as an argument. The `counter` variable is hardwired to 1.

Memory Allocations. Memory allocations are logged on access, when pages are mapped into or revoked from an address space. The `current` domain is derived from the particular address space, which is passed to the log code as argument. The memory allocations are tracked using an array of allocation counters, keyed by accounting domain. The log mechanism retrieves the current allocation counter by indexing into this array, and stores the result in the `counter` variable.

Run-Queue Length. The length of the run-queue is logged periodically, at the timer interrupt. The `counter` variable holds the current length. The run-queue length is a global, per-processor parameter, and no domain or resource principal is involved.

Logging the Data

The log handler finally writes the accumulated data out into the log memory. We have implemented both C-versions and an assembler versions of the handler; the assembler versions are used for instrumenting the IPC fast path. The handler is shown as pseudo-code in Figure 4.7. It basically parses the flags in the log control register, accumulates all missing entries (i.e., those that were not passed as arguments) and writes them out to memory. All log files are cyclic buffers, and there is no notification mechanism that tells the scheduler that a log buffer is full.

For efficiency reasons, the log buffer sizes are powers of two. Further, each log entry row must fit into the buffer without an overlap. The scheduler must set the `padding` flag accordingly, to ensure that the size of a log entry divides the complete log size without a remainder. The first constraint allows a more efficient implementation of

the wrap-around, using only bit operations, but no compares or jumps. The second constraint allows the log mechanism to leave out the wrap-around operations for all intermediate entries of a log entry row, since these entries will always fit into the buffer. When calculating the log buffer size from `size_mask`, the log mechanism crops the result using a maximum size mask. This safety check ensures that the log control registers can be safely co-located with the log files, in the shared log area. Safety issues are discussed in detail in Section 4.3.6.

```
void make_log_entry(ctrl_reg, event_id, current_id,
                    correspondent_id, counter)
{

    idx  = ctrl_reg.current_offset;
    size = (1 << ctrl_reg.size_mask) & max_size_mask;

    if (ctrl_reg.event == TRUE)
        log_area[idx++] = event_id;

    if (ctrl_reg.current == TRUE)
        log_area[idx++] = current_id;

    if (ctrl_reg.correspondent == TRUE)
        log_area[idx++] = correspondent_id;

    if (ctrl_reg.counter == TRUE)
        log_area[idx++] = counter;

    switch (ctrl_reg.timestamp)
    {
        case TS_RDTSC:
        {
            log_area[idx++] = rdtsc();
            break;
        }
        case ...
        ...
     }

    idx += ctrl_reg.padding;
    idx = (idx & ~(size - 1)) | ((idx + 1) & (size - 1));
    ctrl_reg.current_offset = idx;
}
```

Figure 4.7: The log handler in C-like pseudo-code. The handler parses the flags in the log control register, accumulates all entries that were not passed as arguments, and writes the entries out to memory.

4.3.4 Accounting Domains

The following section details how we implemented the accounting domain concept in L4. Like described in Section 3.5, the benefits of the accounting domain concept is ac-

tually twofold. Firstly, accounting domains are managed at user-level, by the scheduler application. By mapping other kernel principals to accounting domains, the scheduler can *specify* itself the particular principal the kernel uses when logging access to resources. Secondly, domains also act as pools or groups of other kernel principals. They can thus be used to *reduce* the logging frequency and the amount of logged data, which ensures that the logging infrastructure can accumulate the workload in an efficient way. Implementing the accounting domain concept in L4 required the following tasks to be done: first, the kernel needs an internal representation of accounting domains. Second, the kernel must provide an interface for management of accounting domains from user-level. The following subsections will discuss both issues in detail:

Internal Representation

In L4, each accounting domain is identified by a 32-bit domain number. For each domain in use, the kernel maintains a data structure called domain descriptor that is used to store domain specific variables. The descriptor is depicted in Figure 4.8 and basically consists of two member variables: (i) an array of per-event log control register selectors, (ii) an array of per-event threshold variables. The log control selector array is used by the logging mechanism, to retrieve the log control register corresponding to a given accounting domain and event. The thresholds are used to implement event sampling.

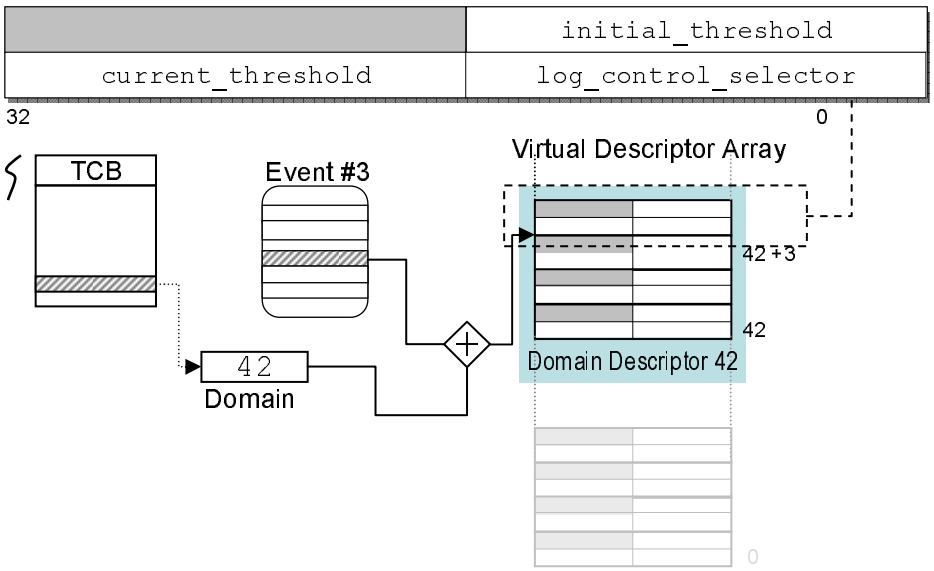

Figure 4.8: Accounting domain implementation in L4. Each domain is represented by a domain descriptor that is held in virtual memory. Threads and address-spaces are mapped to domains via pointer indirection.

Besides the domain descriptor data structures, the kernel needs operations to map principals to their respective accounting domains. The considered workload characteristics require two different kernel entities to be mapped to domains: threads, and address-spaces. Threads are the principals with respect to processor time, performance counters, and IPC usage. Address-spaces are the principals for memory page allocations. To map threads to domains in L4, we enhanced the thread control block with

a member variable containing the thread's accounting domain number. Likewise, we added the domain number to the address-space management data structure, which originally contains other address-space specific data such as page directories and tables.

All descriptors are held in an array keyed by the accounting domain number. To map a given event and principal pair to its corresponding log file, the log mechanism first obtains the principal's accounting domain number from the thread control block or address-space management structure. It then indexes into the descriptor array, to retrieve the particular domain descriptor. Finally, it uses the event identification number to retrieve the log control selector pointing to the control register.

User-Level Interface

To provide an interface for accounting domain management from user-level, we enhanced L4's system call interface with domain-related operations. L4 already provides system calls for thread and address-space manipulation, `ThreadControl()`, and `SpaceControl()`. We enhanced both calls with a `domain` parameter. When a user-level program invokes `ThreadControl()` to create or modify a thread, the kernel modifies the domain member variable in the thread control block according to the `domain` parameter. Likewise, on invocation of `SpaceControl()`, the kernel changes the domain number in the address-space data structure according to the passed `domain` parameter. In L4, operations on threads and address-spaces are restricted to *privileged threads*, such as initial root servers. If the scheduler application is not privileged itself, it must contact a privileged thread and request it to execute `ThreadControl()` and `SpaceControl()` on its behalf.

4.3.5 Data Aggregation Techniques

The logging infrastructure provides several mechanisms that allow scheduler applications to reduce both frequency of logging and the amount of logged data. The following subsection details how we implemented these mechanisms within the L4 microkernel. Besides cyclic logging and event counting, which are already implemented by the log mechanism, the infrastructure features three other mechanisms to aggregate and reduce log data. First, log files can be clustered on a per-event, per-domain base. Second, characteristics that are internal to an accounting domain can be filtered out. Third, scheduling characteristics can be counted in a single log entry instead of being logged in separate log entries. Fourth and last, log events can be sampled, using per-event and per-domain thresholds.

Clustering

Log clustering is used to reduce the amount of data logged on each log operation. The log infrastructure allows schedulers to specify the designated log buffers on a per-event, per-domain base. With buffers local to a domain or an event, the respective identifiers can be omitted, which saves log space. Clustering also relieves the scheduler from searching a log for relevant entries.

To implement this scheme, we map the accounting descriptors writable into the address spaces of privileged L4 threads. The privileged threads can then specify a distinct log control register for each domain and event, by modifying the log control register selector in the domain descriptor. In case the scheduler is not privileged itself, the privileged threads can use L4's memory management primitives to map the domain

descriptors into the scheduler's address space. Section 4.3.6 discusses, how we ensure that write accesses do not have implications on the system's safety.

Domain-based Filtering

Accounting domains can group together other kernel principals. More concrete, schedulers can specify the way how a thread or address-space is mapped to an accounting domain. The domain thereby acts as a "filter" for log events, and the kernel ignores all events that reflect state changes *internal* to an accounting domain.

To implement the domain-based pooling concept, we installed a predicate check on each log event, which checks if the particular event involves an inter-domain state change, or if the change is domain-local . For context-switches, we can simply check if the newly dispatched thread resides in the same domain as the preempted one. For IPC usage, we check if the communication partner is in the same domain. Finally, memory mapping and revocation operations are only considered as relevant for logging, if the two involved address-spaces reside in different domains. Allocation counters are therefore maintained per accounting-domain, not per address-space.

Sampling

The last technique enables schedulers to log only a sample of the scheduling information rather than all. Sampling is based on per-event, per-domain threshold examined before logging the particular information. The schedulers can set these threshold values according to their own specific knowledge of the frequency of log events.

To implement the threshold functionality, we maintain an array of per-event thresholds in each accounting domain descriptor. Since descriptors are mapped writable, schedulers can modify threshold values directly in memory. The thresholds basically consist of two member variables, a counter, and an initial value. The variables are maintained in reverse logic: on each putative log operation, the current counter is incremented and afterwards compared against zero. If so, it is reinitialized with the (negative) initial value, and the workload data is accumulated and logged. Otherwise, the log mechanism directly returns control to the invoking function.

For entry/exit-type events, the threshold functionality is split among the two events. The thresholds are incremented at the entry-event, but compared against zero and reinitialized at the corresponding exit event. Also, while the actual log operation depends on the threshold of the exiting domain, the internal buffers variables must be updated based on the threshold of the entering domain. For context-switches this implies, that two different thresholds have to be checked.

4.3.6 Log Data Access

The memory backing log data and meta-data is directly shared between microkernel and the user-level scheduler application. The kernel references this memory when logging scheduling data. The scheduler application later on evaluates the data by reading it out of the same memory buffers. Additionally, the scheduler is responsible for programming the control data structures initially. The following two subsections illustrate how we implemented access to log data within the kernel, and how the buffers are shared with the user-level. We finally discuss the question of memory provision for log data and meta-data.

Kernel-Level Access to Log Space

For scalability reasons, the kernel holds all log data and meta-data in processor-local memory. This ensures that accesses to log memory on different processors are independent from each other. Since there is only one kernel instance per processor, we did not need to implement mechanisms to synchronize multiple log data producers within the kernel. The log area is mapped to the same virtual address on each processor, which avoids the necessity for per-processor address calculations. For each processor, the log space consists of the memory pages containing log buffers and control registers, plus the pages backing the accounting domain descriptors. The size of the descriptor memory depends on the number of events and the number of accounting domains. In the current implementation, the descriptors are held in a single 4 KByte page. Each descriptor stores two per-event arrays, one for thresholds (4 bytes per element), and the other for the log control register selectors (2 bytes plus 2 bytes padding per element). With four different event types besides the global one for the run-queue length, this yields a size of 32 byte per descriptor, which limits the number of accounting domains to 128. These limits are only artificial and no design limit; nothing, except potential constraints on virtual or physical memory prevents the provision of more descriptors.

At present, our prototype uses statically allocated memory for the log area. For future versions, we propose to use a dynamic allocation scheme similar to the management of thread control blocks in L4 [42]: the new scheme only *reserves* the log areas in virtual memory, without preallocating physical page frames. When the kernel accesses a log area page the first time, the processor will generate a page fault. The page fault handler then must transparently allocate a memory page frame to the faulting address.

User-Level Access to Log Space

To share the log area with the scheduler application, the kernel simultaneously re-maps the log memory regions into the user address spaces of privileged threads. Using L4's user-level memory management primitives, the privileged threads can then redistribute the log memory to the respective scheduler application. For evaluation of log files, the scheduler needs only read access to the log area. However, when programming the log facility, the scheduler must initialize the control data structures, and therefore also needs write access to the log area. The kernel thus re-maps all log memory regions writable. This however implies, that a malicious or flawed scheduler application may corrupt both log data and meta-data. Since log control structures are interpreted by the kernel, corrupt log meta-data may affect the whole system safety. We address this problem by confining the potential implications on safety to the log area. In other words, while we do not prevent the log area from being garbled, we make sure that corrupt log meta-data has no overall consequences for the rest of the system. We achieve this property by meeting the following three conditions:

(1) The log area is only interpreted for logging purposes.

(2) Address calculations for writing log data do not result in arbitrary addresses.

(3) The state of the log meta-data does not result in undefined behavior of the kernel.

We easily achieve the first condition, by not using the shared log area for any other purpose than for logging, and by only using the log area when logging. We accomplish the second requirement by using only relative address offsets in the control data structures. Thus, address calculations always result in an address within the shared log area.

The only exception arises with the `size_mask` parameter. A corrupt log buffer size can result in a log file exceeding the permitted log area. We address this issue by having the log operation crop the log size by a maximum value, which bounds the potential exceeds to a small scratch region behind the actual log area. By leaving this scratch region empty, we ensure that the second condition holds true in every case. We finally meet the third condition automatically, since the rest of the log meta-data is always in a defined state: any setting of the log control register flags is valid by definition. Likewise, all possible configurations of the thresholds always result in a correct threshold logic.

Log Memory Allocation

Generally speaking, it is the the duty of the scheduler application to provide the backing memory for the log data and meta-data. While we can easily implement this policy for instrumented application components, by using the user-level memory management mechanisms provided by L4, we need a different approach to export the control over the *kernel's* log memory to the scheduler. Previous research has shown that kernel memory resources can be safely exported from L4 to the user-level [31]. However, the lack of an implementation of this approach prevented us from incorporating this solution into our logging facility. Currently, the log memory resources are taken from the kernel-internal memory pools.

4.4 Instrumentation of Applications and Device Drivers

With the distributed nature of our component operating system comes along a diversity of the sources of scheduling-relevant information. While some of these characteristics only accrue within the microkernel, others are only available to the user-level modules of the component operating system. Besides the microkernel, we therefore also investigated applicability of logging concepts to the user-level modules of our scenario. We have successfully instrumented both the guest virtual machines hosting the application workload, and the user-level device driver modules.

Since both application workload and the drivers are based on L4Linux, we could apply the same implementation to both components. Also, we could leverage much of the already existing microkernel logging facility. Rather than to detail the complete user-level implementation, we therefore focus on the *differences* with regard to microkernel.

We could directly adopt the dynamic instrumentation and the basic logging mechanism; also, the data aggregation techniques are not specific to the microkernel, and could be easily adopted. One main difference to the microkernel implementation lies in the scheduling characteristics – they are obviously component-specific. The second difference lies in the accounting domain implementation: within L4, we mapped threads and address-spaces to accounting domains. At user-level, we took a different approach of mapping virtual machines to domains. Finally, we used a different scheme to provide the control and log data interfaces between the scheduler and the instrumented application components. The following three paragraphs will discuss these differences in more detail.

4.4.1 Scheduling Characteristics Accruing at User-Level

Since our environment hosts the application workload in virtual machines, it principally supports a diversity of applications and scenarios. In contrast to the microkernel with its limited functionality, there also exists no set of relevant workload characteristics that covers even a portion of the possible scheduling policies. While it is our belief that most of the desired workload data can in fact be accumulated using logging, we surely cannot apply our facility to all those characteristics. We therefore chose to select two relevant examples that accrue within the user-level components: (i) the length of the L4Linux run-queue, and (ii) usage of the shared memory used for interaction between device drivers and application VMs.

L4Linux Run-Queue Length. The virtual machines hosting applications and driver code are based on L4Linux, which is a paravirtualized Linux adoption. While L4Linux generally bases scheduling on L4-threads – spawning a L4Linux-thread for instance results in the creation of a L4-thread –, parts of the original Linux threading functionality still remain active. Beyond that, the scheduling state of the L4Linux-internal threads does not necessarily coincide with their counterparts in L4. As an example, each L4Linux instance creates a server thread that is activated whenever a user-level thread performs a "system call" into the L4Linux kernel. In other words, system calls are translated into control transfers from the user-level thread to the L4Linux server thread. While processing the system call, the server thread thus executes on behalf of the user program. This model implies, that a Linux thread may be in the state *running*, although the corresponding L4 thread is in the state *blocked*, since the thread is currently requesting services from the Linux kernel.

Knowledge of the length of the L4Linux run-queues can therefore be a valuable source for estimating the real load on a processor. Fortunately, the Linux scheduler already tracks the length of the run queues, in a per-processor variable called `nr_running` [48]. To promote the length to the scheduler application, we simply instrumented the `schedule()` function, thereby logging the number of running tasks whenever L4Linux invokes its own internal scheduling logic.

Driver Memory Usage. As discussed in Section 4.3.1, knowledge on memory usage can be a beneficial information for schedulers. In contrast to the *allocations* of memory, which can be efficiently tracked from within L4, communication based on *shared memory* is effectively oblivious to the kernel. This mainly stems from the fact that the kernel cannot distinguish between actively and passively pages, and has no knowledge on the frequency of accesses to shared pages.

However, user-level applications are often more aware of the usage of shared pages. A striking example of such a conscious communication via shared memory in our environment is the interaction between device drivers and the application workload. The translation modules implementing the virtual device interfaces are based on shared memory used to hold driver data or meta-data such as producer-consumer rings or command queues [41]. Intense interaction between an application and a device driver VM will thus result in frequent accesses to these memory regions.

To track those page access patterns, we instrumented the driver side of these translation modules. To make sure, that the references are monitored *in pace*

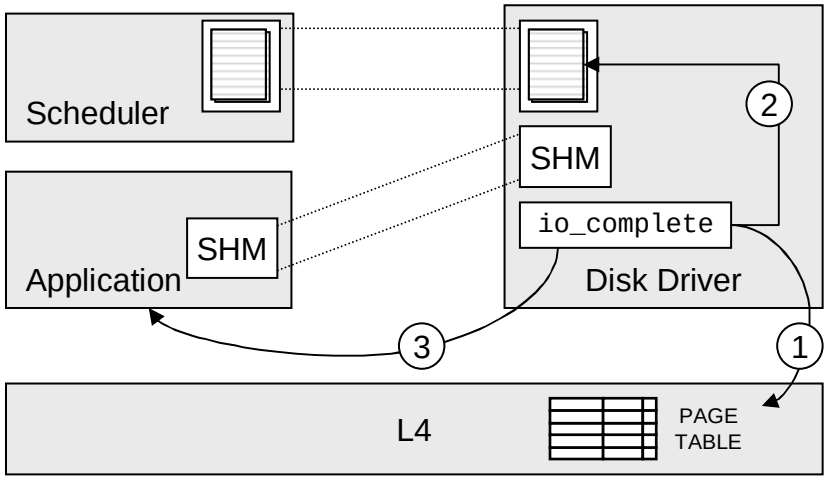

Figure 4.9: Measuring usage of memory shared between applications and the device driver. On completion of driver requests, the log handler retrieves the number of references to shared pages from the kernel and logs the reference counter into a designated log file.

of the normal device driver interaction frequency, we chose to instrument the function invoked upon completion of a driver request. The L4 kernel provides a system call that reports the reference bits of a given memory page. The log code enhancement first retrieves the number of references to the memory pages shared between the driver and the client; it then promotes the reference counter to the scheduler application via logging. Due to the complexity of page table analysis, retrieving page reference bits from L4 is a rather slow operation. Schedulers can adjust the logging thresholds according to the intensity of driver interaction, in order to avoid driver slowdown through the additional instrumentation costs[3].

4.4.2 Accounting Domains

In contrast to the microkernel, which maps threads and address-spaces to accounting domains, we took a different approach for the user-level instrumentation: rather than to base the mappings on component-internal principals, we map *complete virtual machines* to accounting domains; in other words, we consider virtual machines to be the base principals with respect to scheduling. For scheduling on a L4Linux-process level, we would have to incorporate a process-to-domain mapping similar to the scheme used within L4. The scheduler establishes and maintains the mappings from virtual machines to domains; it communicates the mappings to the virtual machine monitor, which in turn passes the respective domain identifier to the kernel whenever it allocates a thread or address-space to a virtual machine. Further, the monitor publishes the mappings to the VMs themselves, by means of a shared memory region.

[3] As an alternative approach, one could to count the number of driver buffers, or commands, instead of the number of referenced pages. This however is a driver-specific approach, and requires knowledge of the internals of a translation module.

4.4.3 Log Control and Data Access

Since the scheduler and the instrumented applications run both on top of L4, we did not need to implement a special interface for interaction with the scheduler. Rather, we could make use of the standard L4 primitives. More concrete, while we needed special system calls for controlling the microkernel's log facility, we could base all user-level interaction on L4's IPC operations. To define the control interfaces, we used an interface definition language, and a compiler that converts the interfaces into optimized assembler stubs [32].

To provide the memory that holds log data and meta-data, we use the hierarchical pager scheme provided by L4 [6]. The scheduler application initially reserves a virtual log memory region for each instrumented L4Linux instance. Whenever a L4Linux instance touches an unmapped log area, the page-fault is forwarded to the corresponding user-level pager, which is the virtual machine monitor. The monitor in turn propagates the fault to the scheduler application, that finally instructs the pager to map the correct page into the address-space of the L4Linux instance. Like with the L4-kernel, there exists only one kernel instance per processor. As we did not instrument any applications programs, the kernel instance remains the only log data producer of a virtual machine on each processor; we therefore did not need to implement mechanisms to synchronize multiple log data producers.

4.5 Implementation of the Scheduler

The following section describes the implementation of the user-level scheduler responsible for allocating processors to the application workload and the drivers. Besides performing the actual allocation policy, which is not part of the logging facility itself and therefore not discussed here, the scheduler must fulfill several tasks in order to obtain the desired characteristics from the instrumented components. More concrete, to make use of the logging infrastructure, the scheduler must perform the following steps: first, it must initialize the log control data structures in the instrumented components (microkernel, device drivers, and applications), to configure and enable the accumulation of the desired workload characteristics. Second, it must parse the generated log data, avoiding however reading of stale or inconsistent data. Third, the scheduler must interpret the particular log data, in case the log data does not yield the desired information directly. The three steps are discussed in the following paragraphs.

4.5.1 Configuring the Log Facility

To program the logging infrastructure, the scheduler first sets up the log control data structures in the shared log area. Each log buffer is reserved by initializing the associated log control register. Afterwards, the scheduler sets the control register selectors in the accounting domain descriptors, to point to the designated log control registers. Also, threshold values are initialized with the appropriate values. Having set all control data structures in the shared log space, the scheduler finally tells the instrumented components to enable logging of the desired workload characteristics. Upon request, the components dynamically instrument the all code events associated with the particular workload characteristics, which finally enables logging.

4.5.2 Parsing Log Buffers

While performance analysis tools usually maintain a global log for all types of log data, our logging facility was explicitly designed to separate different workload characteristics from each other. Consequently, the scheduler is relieved from *selecting* relevant log entries out of a log buffer. It can simply parse linearly through each log buffer and evaluate all log entries. However, two other problems arise when analyzing log files. First, the scheduler must prevent reading *stale* log date that has been analyzed in a previous run. Second, since log buffers are directly shared with the instrumented components, the scheduler must prevent reading *inconsistent* data.

To avoid reading stale log data, the scheduler can employ two different techniques: first, it can use the log time stamps (or other knowledge on the frequency of log operations), to distinguish stale from fresh log entries. Time stamps do not need to reflect real time characteristics, and a virtual time stamp is sufficient. Second, if time stamps are not available (e.g., to save log space) the scheduler can mark all parsed log entries as invalid, by using a magic number. However, the second solution can significantly affect performance due to cache-line bouncing, if the log buffer is located on a different processor. We therefore resort to the first solution.

Reading inconsistent data is avoided by detecting writes to a log buffer that is simultaneously being read by the scheduler. Rather than to implement a solution based on reservations and valid bits [67], we developed a different solution, which relieves the log operation from setting valid bits during logging. The write-ordered memory model of the Pentium processors [36] automatically ensures that the current index always reflects the state of the log file consistently. The scheduler uses the index in order to detect if the log buffer was written concurrently. Before parsing log buffers, the scheduler saves the log file's current index. When the analysis is finished, it first executes an `mfence` instruction that serializes all memory reads prior to this instruction. Afterwards, it verifies that the current index matches the saved copy, by re-reading the index from memory. If they do not match, it restarts the complete analysis process. This strategy fails if the concurrent log data producer generates *exactly* as much log entries as there is room in the log buffer. In this case, the current index remains the same, although the complete log buffer has been replaced meanwhile. To address this corner-case, the scheduler must additionally verify time stamps (or other entries that are known to change), in case the index matches the saved copy.

Our solution may result in a starving analysis process, if the log file is permanently being written during analysis. To address this problem, the scheduler additionally disables the log data procedure in the uncommon case that it detects an index mismatch. A simple technique to disable write-accesses on a per-log-buffer base is to atomically clear all flags in the corresponding log control register. Since the log producer may reside on a different processor, we must make sure that the updated register flags are eventually promoted to the remote processor. On IA-32 processors, this requires the serializing operation `mfence` to be executed on the remote processor. To avoid executing this operation on each log operation, we added it to the periodic timer interrupt handler, which bounds the waiting time of the analysis process to at most one timer tick.

To illustrate the complete log buffer analysis, an example is shown in pseudo-code in Figure 4.10. The example shows the evaluation process for an L4-provided log file. Stale log entry detection is based on time stamps. The time stamp is assumed to be held in the last entry of each row. The analysis procedure first retrieves the domain descriptor (`desc`) corresponding to the particular event type and domain, by scanning

the virtual array of domain descriptors shared between L4 and the scheduler. It then dereferences the log control selector (`selector`) held in the domain descriptor and locates the log control register in the log area (`ctrl_reg`). Log analysis will start immediately *after* the row currently being written (`cur_idx`) and end immediately *before* the current index (`last_idx`); the current entry must be ignored, since it is prone to concurrent writes that do not modify the index. Before starting the actual analysis, the log procedure obtains a local copy of the current index (`chk_idx`), and a copy of the time stamp in the last completely written log entry row (`chk_time`). For each entry row in the log file that has not been evaluated yet, the procedure parses the respective entries. Having processed the complete log buffer, the procedure checks if the current index was modified meanwhile. If not, it additionally checks the saved time stamp, to detect the corner case of a producer having written exactly as much log entries as the buffer can store. If any of the two checks fails, all log control register flags are cleared and the analysis process is restarted. After a successful retrial, the flags are finally reset again. The procedure returns the time stamp of the most recent entry row; it is passed to the analysis procedure in the next run.

4.5.3 Deriving Scheduling Information

Deriving scheduling information from log files is dependent on the particular policy, which is not part of the logging infrastructure itself. To outline how this process may be accomplished, the following section will present a set of example queries that derive three different characteristics: processor load, processor affinity, and finally domain locality information.

Processor Load. To estimate the load on a processor, the log of the particular load gauge (e.g., the run-queue length in L4 or L4Linux) is parsed. Usually, first the individual load on each processor is calculated [8,48]. Individual load values are then compared against each other [53], or against the overall or average system load [26]. Hence, costs and complexity of measuring processor load will grow with the number of processors and the number of entries in the load history.

Processor Affinity. Determining affinity to a processor is done by analyzing log data of processor associated resources, like the processor's caches or the dedicated device driver modules. Processor affinity is a domain property, therefore affinity values must be calculated for each processor and each domain. Many cache-affinity policies only consider the last time a domain or process ran on a processor to be relevant [9, 12, 18]. However, also a set or history of values may be viable. Costs and complexity will correlate with the number of processors, the number of domains in use, and the number of entries in the log buffers.

Domain Locality and Domain Working Sets. Identifying locality relations and domain working sets requires scanning a correspondent log file that records access to shared resources such as the IPC operations provided by L4. In some cases, a domain to be scheduled is already given, and the goal is to calculate the domain's locality to other domains, in order to identify potential "gang members" [12]. But communication patterns can also be used to calculate *process working sets*. That is, complete gangs are identified, not only a set of domains associated with a particular domain [25]. Costs and complexity heavily depend on the scope (global, clustered, or local) of log buffers. Principally, a correspondent history will grow

quadratically with the number of domains, and linearly with the number of entries in the log buffers[4].

[4]Costs of calculating a transitive closure as in [25] are not considered here

```
int parse_l4_logfile(event_id, domain_id, last_timestamp,
                     row_size)
{

#define NEXT_ROW(idx) (idx & ~(log_size - 1)) |\
                      ((idx + row_size) & (log_size - 1))
#define PREV_ROW(idx) (idx & ~(log_size - 1)) |\
                      ((idx - row_size) & (log_size - 1))

    desc        = l4_descriptor[domain_id];
    selector    = desc.log_control_selector[event_id];
    ctrl_reg    = l4_log_area[selector];
    log_size    = 1 << ctrl_reg->size_mask;
    restarted   = false;

restart:
    cur_idx     = NEXT_ROW(ctrl_reg->current_offset);
    cur_time    = last_timestamp;
    last_idx    = ctrl_reg->current_offset;
    chk_idx     = ctrl_reg->current_offset;
    chk_time    = l4_log_area[PREV_ROW(chk_idx) + row_size - 1];

    while (cur_idx != last_idx)
    {

        if (l4_log_area[cur_idx + row_size - 1] > cur_time)
        {
            analyze_log_entry_row(cur_idx);
            cur_time = l4_log_area[cur_idx + row_size -1];
        }
        cur_idx = NEXT_ROW(cur_idx);
    }

    mfence();
    rechk_idx  = ctrl_reg->current_offset;
    rechk_time = l4_log_area[PREV_ROW(chk_idx) + row_size - 1];

    if (chk_idx  !=  rechck_idx ||
        chk_time !=  rechck_time)
    {
        atomic_clear_flags(ctrl_reg);
        restarted = true;
        goto restart;
    }

    if (restarted == TRUE)
        atomic_set_flags(ctrl_reg);

    return cur_time;
}
```

Figure 4.10: Example log analysis procedure in C-like pseudo-code. The example shows the evaluation process for an L4-provided log file of size `log_size`, with `row_size` entries per row. The last entry in each row is assumed to store a time stamp.

Chapter 5

Evaluation

The goal of this work is to support the flexible and extensible design of user-level schedulers, by providing a efficient and scalable way to accumulate and promote scheduler-relevant information. We conducted several experiments to evaluate if our approach achieves the desired qualities in terms of efficiency and scalability. We considered two different aspects as relevant, the *baseline costs*, and the *overall effects on application performance* of log data accumulation and analysis. In this chapter, we present the performed experiments and discuss the results.

5.1 Experimental Environment

In all experiments, we used a current version of L4Ka::Pistachio, the latest L4 microkernel developed at the University of Karlsruhe. The L4Ka virtualization environment hosting applications and user-level device drivers is based on two different generations of the Linux kernel: version 2.4, and version 2.6. We chose the latter version for our evaluation. The virtualization code base also provides the interface modules for providing driver services to the applications. We conducted the measurements on an *IBM eServer xSeries 445*, with 8 hyper-threaded Intel Xeon CPUs. Each of the CPUs has a clock speed of 2.2 GHz, 12 KBytes I-Cache and 8 KBytes D-Cache, 512 KBytes level 2 unified cache, and 2048 KBytes level 3 unified cache. The xSeries 445 implements a NUMA-based, cache-coherent architecture, with 2 nodes à 4 CPUs and 2 GBytes node-local memory.

5.2 Baseline Log Performance

In order to determine the efficiency of logging, we measured the performance of logging at the most critical path in the L4 microkernel: the IPC system call. Performance of the IPC call is a metric traditionally used for L4-based systems [33,42]. Since several scheduler-relevant events are on the critical IPC path – thread switches for measuring timing or cache properties, and the software event used to log IPC communication patterns – fast path performance is also highly revealing for measuring log performance.

5.2.1 Logging Overhead

For measurement of CPU overhead, we compared a native version of L4Ka::Pistachio, which does not support event-logging, against our modified version, which provides logging support. For all baseline measurements, we used a log file size of 512 bytes. We enabled logging only of IPC usage, and disabled all other characteristics. As described in Section 4.3.1, we normally log IPC usage twice, once for the sender and once for the receiver; however, since we wanted to measure the base performance of *one single* log operation, we disabled the second operation. Also, by logging IPC usage, we disregard any overhead for accumulating counters, since this overhead depends on the particular characteristic. If the overhead is constant – which holds true for all characteristics recorded on the fast path – one can easily offset it to the results.

To create "IPC load", we used the `pingpong` benchmark, which basically consists of two threads on a processor that continuously send short IPC messages to each other. The benchmark measures the round trip time in clock cycles for message sizes from 0 to 252 Bytes. Since transfer times are very short, the benchmarks conveys messages repeatedly for each size. Originally, `pingpong` deploys only one pair of threads that both run on a single CPU. To evaluate if the logging facility achieves good *multiprocessor* performance and keeps separate log data producers independent, we developed a parallel version of the `pingpong` benchmark. The parallel version creates eight pairs of threads, with each pair each dedicated to one of the Xeon CPUs[1]. It then lets all thread pairs run concurrently on their respective CPUs, which results in eight processor-local, but simultaneous sequences of message transfers. Like with the original uniprocessor benchmark, each thread pair measures round trip time per message size; the main thread finally collects all round trip results from the different processors. To evaluate logging performance, we further had to add support for placing the threads pairs into different accounting domains. We performed all round trip measurements both for IPCs within the same address-space (Intra-AS) and across address-spaces (Cross-AS). We compared the following configurations:

"native". The original, unmodified kernel version without support for logging.

"disabled". The instrumented version, but with disabled logging. This configuration matches the case where the kernel provides support for logging, but no log event is currently enabled by the scheduler. It also matches the case where a given event is internal to a specific domain, and therefore not considered for logging; this predicate is always checked, no matter if a given event is turned on or off[2].

"0-entry". In this configuration, logging is enabled, but all log entry flags are cleared. Thus, the processor transfers control to the log mechanism and performs the required threshold checks and address calculations; but it does not actually write an entry.

"1-entry" and "4-entry". Equals the previous configuration, but this time actual log entries are written. In the case of "1-entry", the log mechanism only logs the

[1] To avoid hardware interferences, we did not make use of the hyper-threading features, and only used one virtual CPU on each physical CPU.

[2] At present, we do not instrument the domain checks dynamically. A given code path may be instrumented more than once – like the context-switch, which is instrumented for processor service time, performance counter information, and IPC usage. If we dynamically inserted the checks on such code paths, we either would have to execute the predicate checks multiple times, or alternatively implement some functionality to check which events are actually enabled. Since the overhead of a domain check is minuscule, we chose to skip dynamic instrumentation of the domain checks completely.

event identifier; in the case of "4-entry", it logs event, current, and correspondent identifiers, and the counter, which is hard-coded to 1 for IPC usage.

"5-entry". Equals the "4-entry" configuration, but additionally logs a time stamp based on the hardware time stamp counter register. The log handler also inserts a padding of 3 entries, in order to round up the number of entries to a power of two.

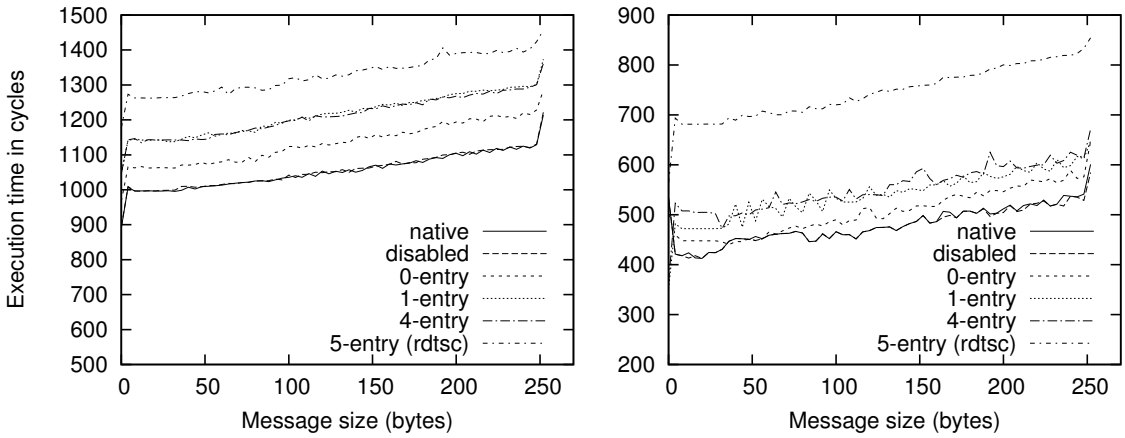

Figure 5.1: Costs of a log operation on the IPC path, across address-spaces (left), and intra address-space (right).

Figure 5.1 depicts the performance of the different configurations, averaged over the processors. A further summarized version of these results is given in Figure 5.2, which lists the overhead averaged over processors and message sizes, in absolute clock cycles and in percent of the native performance. Disabled logging does not have any overhead. For logging zero to four entries, the overhead is less than 16 percent for cross and intra address-space transfers. The most expensive log operation, with five log entries and about 26 respectively 55 percent penalty, adds an extra overhead to the performance, which is caused by the `rdtsc` instruction to read the time stamp counter[3]. The differences in the number of instructions between cross address-space and intra address-space transfers probably arise from the instruction cache, which is flushed on a reload of the page table base register.

		native	disabled	0-entry	1-entry	4-entry	5-entry
Cross AS	cycles	(1053)	2	80	161	157	275
	percent	(0.0)	0.2	7.6	15.4	15.0	26.2
Intra AS	cycles	(480)	-1	26	59	72	264
	percent	(0.0)	-0.3	5.6	12.5	15.1	55.1

Figure 5.2: Average overhead of log operations on the IPC path, relative to native performance.

[3] As a side note, on an Opteron-based processor, the `rdtsc` operation costs only about 6 clock cycles, which renders it an accurate *and* efficient way to measure timing properties.

In general, the results are very promising; in absolute terms, logging overhead is minuscule; for less critical instrumentation points, the overall effect will certainly be negligible. Specifically on the fast path, performance drops about 10 to 15 percent in most cases, and about 55 percent in the absolute worst case. However, the flexible design of the logging facility leaves plenty of room to reduce this overhead: First, the scheduler can disable irrelevant log events, which reduces the overhead to zero. Second, it can leverage the accounting domain concept, to filter out interactions between equally accounted threads. Third, it can employ the threshold functionality, which will reduce the overhead linearly with the value of the thresholds.

To illustrate the scalability of our logging facility, we have depicted the results in a slightly different form in Figure 5.3. The figure draws the round trip time *per processor*, averaged over the message sizes. The curves clearly demonstrate that log data producers on different processors are completely independent from each other, and no interference occurs.

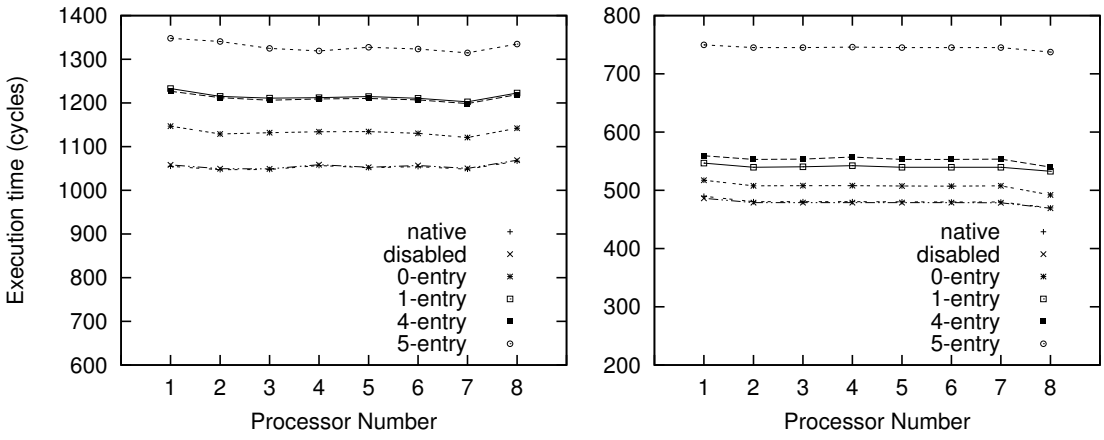

Figure 5.3: Performance of log operations on different processors, across address-spaces (left), and intra address-space (right).

5.2.2 Log Analysis Overhead

In addition to evaluating the performance of log data generation, we also measured the overhead of log data analysis. In general, log data analysis overhead will grow linearly with the amount of memory being read; however, ensuring log data integrity may incur additional overhead, since it implies that the analysis process is restarted in the case of concurrency. To evaluate the effects induced by the restarts, we used the following scenario: we constantly produced log data at a high frequency, by enabling a log event, which occurs very frequently in a L4 based system – the IPC log event on the fast path. We simultaneously ran an analysis process that tries to read the generated log data. Although this scenario is rather artificial, and unlikely to occur in a real-world environment, it still helps us to evaluate the *worst case* overhead of log analysis.

To cause the kernel to log IPC usage as fast as possible, we used the canonical `pingpong` benchmark (we used intra address-space transfers). This time, we modified the benchmark such that the thread pairs continue exchanging messages *ad infinitum* rather than to stop eventually and measure round trip time. We then developed a simple

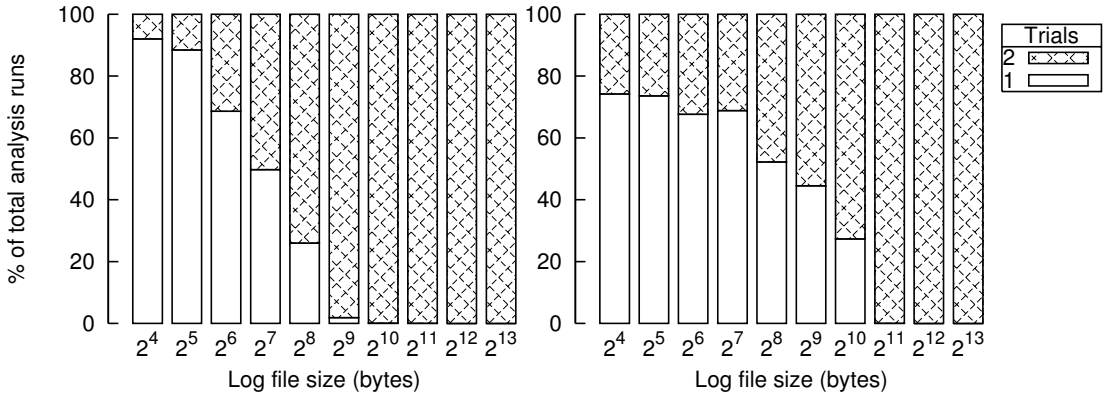

Figure 5.4: Analysis retrials, with clearing log control register flags. The node-local case is shown on the left hand, the node-remote case on the right hand.

scheduler application that continuously tries to read the log buffer being written by the kernel. On each message transfer, the kernel logs two entries, the correspondent identifier, and a virtual time stamp. The scheduler implementation follows the algorithm described in Section 4.5: it first tries to read the complete log buffer, and afterwards verifies that the buffer was not written meanwhile, by re-checking the current index in the log control register, and the time stamp of the log entry row before the current one. The scheduler restarts the procedure, until it finally succeeds in reading the complete log buffer uninterruptedly. To evaluate our approach of disabling logging remotely if the check fails, we compared two different configurations of the scheduler application: in the first configuration, the scheduler clears the control register flags before restarting the analysis; in the second, it only restarts the analysis. Since L4 is not preemptible, the analysis process will always succeed in the first run, when reading processor-local data. However, if the log buffers reside on a different processor, there is a high chance that the remote kernel instance produces new log data while the scheduler is reading the buffers. We measured the number of trials the analysis process requires to read remote log buffers of different sizes. For each size, the scheduler performs 100,000 consecutive analysis operations, and afterwards prints out the distribution of the number of retrials required for each operation. We conducted the measurements for reading *node-local* respectively *node-remote memory*, that is, with the scheduler and the `pingpong` threads residing within the same node respectively in different ones.

The results are depicted by Figures 5.4 and 5.5. The figures draw the distribution of the number of trials, for log file sizes ranging from 16 bytes up to to 8192 bytes. The number of trials generally increases with the size of the log file. We consider the hardware cache-coherency protocols (and the different overheads for node-local and node-remote memory) to be responsible for the variances in the distribution[4]. Figure 5.4 shows the number of trials in the first configuration, in which the scheduler clears the control register flags. In this case, the number of analysis trials is limited to 2. For log file sizes of more than 2048 bytes, the analysis process always requires two

[4]We ran the same setup on a 2-way AMD Opteron multiprocessor, which uses a simple bus-snooping protocol to keep the caches coherent. On this architecture, the variances disappeared in both configurations, and the number of trials increased linearly with the size.

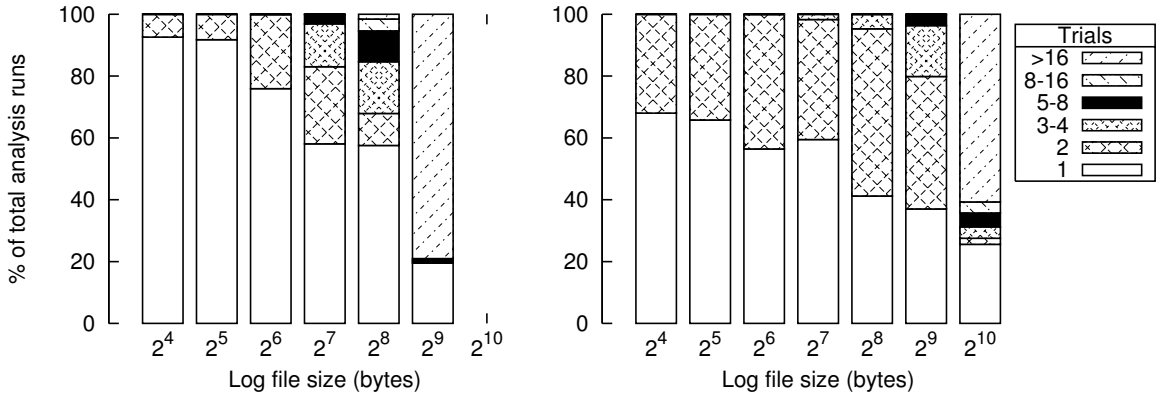

Figure 5.5: Analysis retrials, without clearing log control register flags. The node-local case is shown on the left hand, the node-remote case on the right hand.

runs for a successful reading. Figure 5.5 shows the second configuration, in which the process simply retries reading the buffers, without clearing the control register flags. One can see, that this is a practical approach for smaller log file sizes; in this case the chance is high that consecutive analysis trials will eventually go through. However, with increasing log file size, the analysis process requires more parsing time, which results in a decreasing analysis frequency and a higher chance of interruption by the log data producer. The number of trials grows exponentially with the approximation of log analysis and log producing frequency. As soon the frequencies match, the analysis will starve forever. We canceled the measurements for sizes higher than 2048 Bytes, as they required absurdly much time.

Figure 5.6 shows the CPU overhead for log analysis, in a double-logarithmic scale. While the curves of the first configuration grow more or less linearly with respect to the log buffer size, the curves of the second configuration grow exponentially. The lowermost curve shows the performance of log analysis in the processor-local case. To summarize, with the proper analysis algorithm, the costs of parsing a log buffer are completely memory-bound, and ensuring data integrity using requires only a small and constant overhead.

5.3 Application-Level Performance

To evaluate the overall effects of logging on application-performance, we implemented an example scheduler application responsible for allocation of processors to virtual machines. The policy implements a basic affinity scheduling which relies on *interaction patterns* between application workload and device drivers (the drivers associated with a particular CPU). The interaction patterns are gathered in two different ways: first, by analyzing the IPC usage logs provided by L4; second, by analyzing the shared memory reference logs provided by the device drivers. Using data from multiple sources not only allows us to evaluate the additional costs of *accumulating* log data. Logging two characteristics with a similar meaning also demonstrates that our facility can be used to investigate the *expressiveness* of scheduling characteristics.

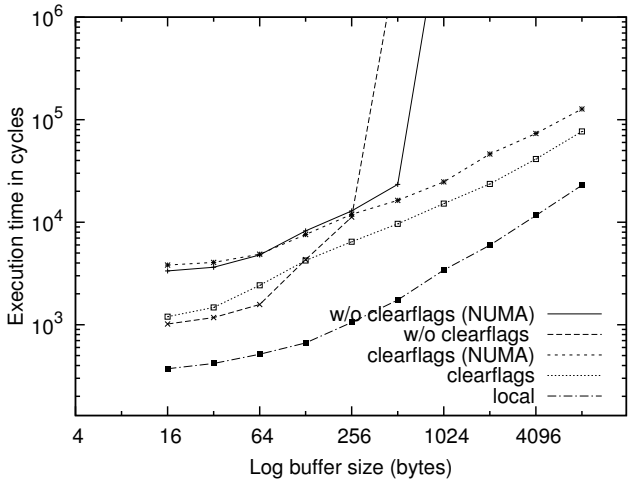

Figure 5.6: Analysis overhead in cycles, with and without clearing the log control register flags, both for reading node-local, and node-remote memory. The lowermost curve shows the analysis overhead in the processor-local case.

To monitor IPC and shared memory usage, the scheduler initially programs the log facilities, using the control interface; it further associates a separate accounting domain number with each virtual machine instance, and communicates the associations to the virtual machine monitor. The monitor passes the respective domain number to the kernel whenever it allocates a thread or address-space to a virtual machine. The monitor further publishes the associations to the virtual machines themselves, in a dedicated shared memory region. The kernel is configured to use domain-local IPC buffers, and to store the correspondent domain and a time stamp on each IPC. The device drivers use a global log buffer for the memory references. The drivers accumulate the references upon completion of a driver request, by calling into the kernel and retrieving the particular page reference bits.

To calculate the actual affinity values, the scheduler maintains a per-driver data structure that holds the scheduling statistics derived from the logs. Each data structure basically consists of two counter vectors, one to count the IPC interactions with the driver's clients, and the other to count the references to the memory shared with the clients. To calculate the two counters from the logs, the scheduler deploys a separate thread, which periodically wakes up and parses the IPC and shared memory log buffers. For each log entry row, it increments the per-domain correspondent counters appropriately. The counter vectors finally yield a matrix of client-to-driver transactions, reflecting the affinity between applications and device drivers.

We evaluated this scenario in three aspects: first, we calculated the amount of memory that is required to accumulate IPC and shared memory usage. Second, we compared the resulting device driver throughput against a native version of the scenario, which completely lacks support for logging. Third, we evaluated the expressiveness of the two scheduling characteristics. For measurements, we used a device driver providing access to a hard disk, and a client that accesses this hard disk using the driver's virtual disk interface. In order to eliminate all I/O processing costs, we let the device driver export a *RAM disk* of size 256 MByte, rather than a real hard disk. The driver interface is based on a shared memory segment comprising four memory pages; the log

code accumulates the reference counters of these four pages upon completion of each block request. To generate disk load, we ran the Postmark benchmark in the client, on the RAM disk exported by the device driver. Postmark simulates the workload of an Internet electronic mail server, by creating a large pool of continuously changing files. We configured Postmark to use files ranging between 500 bytes and 1 MByte in size, a working set of 200 files, and 10000 file transactions.

5.3.1 Memory Requirements

To determine the amount of log data memory required for this scenario, we first performed a dry-run of the benchmark, in order to obtain the number of IPC and shared memory transactions that take place during each analysis interval. The size of the log files must correspond with these numbers, otherwise the periodic scheduler thread will loose some of the usage statistics. To count the usage numbers, we used the *event counter* mechanism provided by the logging facility. In the case of shared memory, we counted the number of log *invocations* rather than the number of references. We set the scheduler wakeup interval to 20 milliseconds, and took 500 samples of IPC and memory usage counters while the disk driver was handling the Postmark benchmark. The samples yield average numbers of 29 IPCs and 437 shared memory log invocations, and maximum numbers of 90 IPCs and 973 shared memory log invocations per scheduling interval[5]. Hence, the resulting memory requirements add up to at least 512 Bytes for each per-domain IPC log file, and to 4096 Bytes for the global device driver log file. We used this configuration for the actual benchmarks.

The current prototype implementation employs a 64 KByte log space area per instrumented component and processor. The evaluation shows, that this is enough for few domains and global logs, but may be problematic especially with an increasing number of accounting domains or domain-local logs. For future versions, it may be required to reserve more space to hold the log buffers.

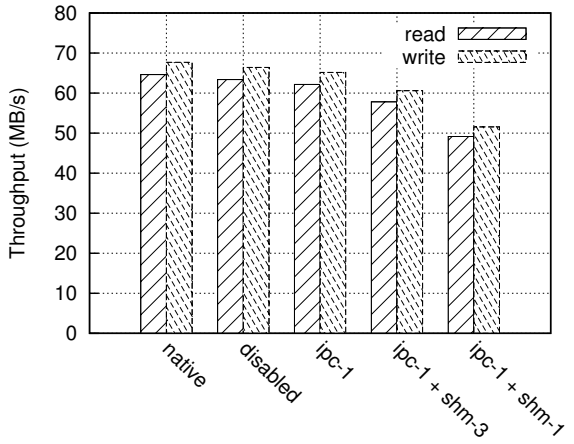

Figure 5.7: Postmark throughput on a RAM disk, for different log configurations.

[5]The disk driver implementation uses interrupt coalescing, therefore the number of IPCs is much lower than the number of completed block requests.

5.3.2 Device Driver Throughput

To measure the overall effects on device driver throughput, we obtained the results of the Postmark benchmark for five different configurations: the first configuration runs without any support for logging, that is, neither the instrumentation code nor the scheduler application is enabled. In the second configuration, logging is supported but disabled; the scheduler application is periodically invoked but performs its calculations on empty buffers. In the third configuration, only IPC usage is gathered. The fourth configuration monitors both IPC and shared memory usage, but accumulates the latter only on every third event, using a log threshold of 3. Finally, the last configuration accumulates both IPC and shared memory usage on every single event.

The results are shown in Figure 5.7. For configurations that do not monitor memory references, the overall effects are not dramatic. For real device drivers, which do involve I/O-processing, the effects will be even lower than in our memory-bound scenario. Disabled logging but enabled log data analysis reduces throughput to about 98 percent compared to the native setup. Accumulating IPC usage reduces throughput to 96.2 percent relative to the native setup. As expected, monitoring shared memory usage is an expensive task. Logging shared memory on every third event causes the performance to drop to about 89.5 percent compared to the native setup; accumulating on every single event yields the worst result, with a throughput of about 76.1 percent.

5.3.3 Correlation of IPC and Shared Memory Usage

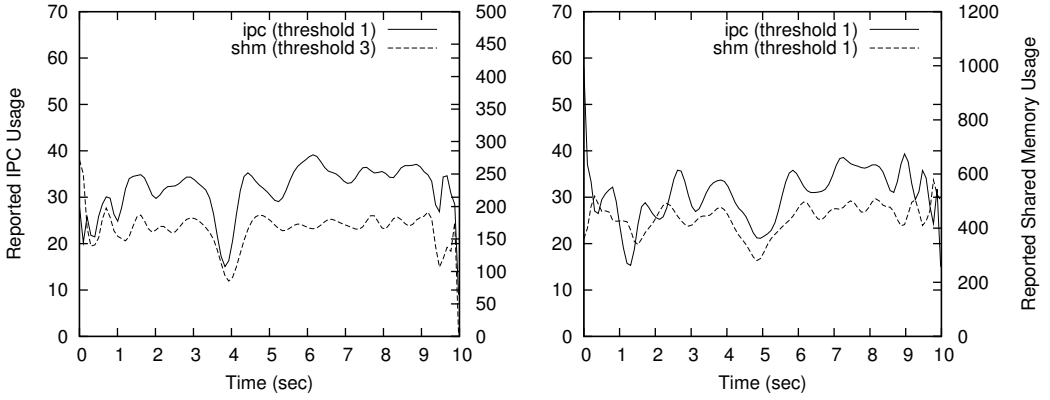

Figure 5.8: Correlation of IPC usage and shared memory references during the Postmark benchmark, for different log thresholds.

Investigating adequateness or meaningfulness of different scheduling characteristics is certainly *not* a goal of our work; however, what we *can* demonstrate is, that the logging facility may be a helpful tool for such an investigation. We therefore evaluated the expressiveness of the accumulated scheduling data, by comparing IPC and shared memory usage counters against each other. For this purpose, we modified the scheduler application to sample the usage counters at 500 subsequent scheduler invocations, which resulted in a distribution of IPC and memory reference counters over a period of 10 seconds. All samples were taken while the disk driver was handling the Postmark benchmark. To obtain a trend of the distribution, we smoothed the resulting data sets using the Bézier smoothing method of `gnuplot`. The curves are shown in Figure 5.8.

The graph on the left side shows the configuration, in which shared memory usage is accumulated on every third event; the graph on the right hand depicts the configuration with single-event accumulation of shared memory. Although the figures only show a trend line, one can still observe that, at large, IPC and shared memory usage correspond to each other. Furthermore, there is no substantial difference between logging memory references on every single and every third event.

Chapter 6

Conclusion

The goal of this thesis is to overcome the limitations and inflexibilities of current approaches with respect to scheduling-relevant information. To achieve this objective, this thesis introduces a system based on instrumentation and event-logging, which is able to propagate the required information to the user-level scheduler in a flexible, efficient, safe, and scalable manner.

Existing approaches to support multiprocessor scheduling all have their advantages and limitations, but none of them achieves all of the desired requirements: some approaches have direct and fast access to scheduling information, but are overly limited with respect to the offered policies, or with respect to their scope. Others provide flexible support for performing scheduling decisions, but either they exhibit poor performance in cases of frequent scheduling events, or they are limited to specific system components such as the operating system kernel. Finally, there exist approaches that allow for user-controlled scheduling and for direct access to scheduling information, but jeopardize the system's safety for it.

This thesis introduces a new mechanism that achieves all of the required properties. Based on system instrumentation and event-logging techniques, the mechanism provides a flexible and scalable way to accumulate scheduling characteristics at run-time, from different components in the system. Suitable to heap large amounts of scheduling data, logging remains efficient even in cases of frequently changing scheduling characteristics. Designed with safety requirements in mind, the mechanism is not only applicable to user-level applications, but also to crucial system components such as the privileged operating system kernel.

We see our work as a starting point to evaluate existing, or to develop new scheduling extensions at user-level. It is our belief that many existing policies can easily be ported and adapted to make use of our logging infrastructure. Although we have investigated different multiprocessor scheduling policies, and also developed some experimental implementations in order to evaluate our infrastructure, we were unable to consider *all* of them during evaluation. Implementing new policies would therefore be helpful for our infrastructure as well, since it would provide valuable feedback and offer the chance for further improvements and enhancements. Aside from supporting scheduling policies, our infrastructure could also act as a tool to *evaluate* scheduling characteristics, with regard for instance to their meaningfulness, or the costs of their accumulation. Furthermore, while the focus of our approach has been on support for multiprocessor scheduling, we see no particular reason why it should not be applicable to other scheduling problems as well, as for instance, real-time scheduling. How-

ever, other scheduling problems may have different constraints, and care must be taken when applying our scheme there. For instance, the hard timing requirements of real-time scheduling may demand for *both* asynchronous and synchronous mechanisms to promote scheduling information, since real-time schedulers usually rely on accurate delivery of scheduling events. Finally, our work supports user-level scheduling at the input side, but relies on existing mechanisms at the output side, such as thread dispatching or migration routines provided by the operating system. Future work has to be done to investigate if the mechanisms on the output side are adequate for use by our facility.

Bibliography

[1] Daniel Nussbaum Alexandra Fedorova, Christopher Small and Margo Seltzer. Chip multithreading systems need a new operating system scheduler. In *Proceedings of the eleventh ACM SIGOPS European Workshop*, September 2004.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the sixteenth ACM Symposium on Operating systems principles (SOSP)*, pages 357–390. ACM Press, October 1997.

[3] Thomas Anderson, Edward Lazowska, and Henry Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. In *Proceedings of the 1989 ACM conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 49–60. ACM Press, May 1989.

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the thirteenth ACM Symposium on Operating systems principles (SOSP)*, pages 95–109. ACM Press, October 1991.

[5] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michael Ostrowski, Bryan Rosenburg, Robert Wisniewski, and Jimi Xenidis. Scheduling in K42. White Paper, August 2002.

[6] Mohit Aron, Jochen Liedtke, Kevin Elphinstone, Yoonho Park, Trent Jaeger, and Luke Deller. The sawmill framework for virtual memory diversity. In *Proceedings of the sixth Australasian conference on Computer systems architecture (ACSAC)*, pages 3–10. IEEE Computer Society, December 2001.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the fourth USENIX Symposium on Operating systems design and implementation (OSDI)*, pages 45–58. USENIX Association, February 1999.

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

[9] James M. Barton and Nawaf Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *Proceedings of the 1995 IPPS workshop on Job scheduling strategies for parallel processing (IPPS)*, pages 24–40. Springer-Verlag, April 1995.

[10] Anan Batat and Dror. G. Feitelson. Gang scheduling with memory considerations. In *Proceedings of the fourteenth International Parallel and distributed processing symposium (IPDPS-00)*, pages 109–114. IEEE Computer Society, May 2000.

[11] Frank Bellosa. Follow-on scheduling: Using TLB information to reduce cache misses. In *Proceedings of the sixteenth ACM Symposium on Operating systems principles (SOSP), Work in Progress Session*. ACM Press, October 1997.

[12] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113–121, August 1996.

[13] Brian N. Bershad, Craig Chambers, David Becker, Emin Gun Sirer, Marc Fiuczynski, Stefan Savage, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles (SOSP)*, pages 267–284. ACM Press, December 1995.

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

[15] Tim Brecht and Kaushik Guha. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27/28(4):519–539, October 1996.

[16] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX annual technical conference)*, pages 15–28. USENIX Association, June 2004.

[17] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[18] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 12–24. ACM Press, October 1994.

[19] Benjie Chen. Multiprocessing with the exokernel operating system. Master's thesis, Massachusetts Institute of Technology, Deptartment of Electrical Engineering and Computer Science, December 2000.

[20] Peter Druschel, Vivek S. Pai, and Willy Zwaenepoel. Extensible kernels are leading OS research astray. In *Proceedings of the sixth IEEE workshop on Hot Topics in Operating Systems (HotOS)*, pages 38–42. IEEE Computer Society Press, May 1997.

[21] Kevin Elphinstone. Future directions in the evolution of the L4 microkernel. In *Proceedings of the NICTA workshop on OS verification 2004*. National ICT Australia, October 2004.

[22] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles (SOSP)*, pages 251–266. ACM Press, December 1995.

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

[24] Dror G. Feitelson and Larry Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65, May 1990.

[25] 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.

[26] Hubertus Franke, Shailabh Nagar, Mike Kravetz, and Rajan Ravindran. PMQS: Scalable Linux scheduling for high end servers. In *Proceedings of the fifth annual Linux showcase and conference*. USENIX Association, November 2001.

[27] Dipak Ghosal, Giuseppe Serazzi, and Satish K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443, May 1991.

[28] Richard Gibbons. A historical application profiler for use by parallel schedulers. In *Proceedings of the 1997 IPPS Workshop on Job scheduling strategies for parallel processing (IPPS)*, pages 58–77. Springer-Verlag, April 1997.

[29] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. *ACM SIGARCH Computer Architecture News*, 32(5):260–270, 2004.

[30] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of performance on parallel applications. In *Proceedings of the 1991 ACM conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 120–132. ACM Press, May 1991.

[31] Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Proceedings of the eighth Asia-Pacific Computer Systems Architecture Conference (APSEC)*, pages 277–289. Springer-Verlag, September 24–26 2003.

[32] Andreas Haeberlen, Jochen Liedtke, Yoonho Park, Lars Reuther, and Volkmar Uhlig. Stub-Code performance is becoming important. In *First Workshop on Industrial Experiences with Systems Software (WIESS-00)*, pages 31–38. USENIX Association, October 2000.

[33] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-based systems. In *Proceedings of the sixteenth ACM Symposium on Operating systems principles (SOSP)*, pages 66–77. ACM Press, October 1997.

[34] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 187–197. ACM Press, October 1992.

[35] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Goncalves, Oscar Naim, Zhichen Xu, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT*, pages 201–213. IEEE Computer Society, September 1997.

[36] Intel Corporation. *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 1999.

[37] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Ltd., 1st edition, 1991.

[38] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, February 1995.

[39] Thomas Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[40] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 226–36. ACM Press, May 1990.

[41] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the sixth USENIX Symposium on Operating systems design and implementation (OSDI)*. USENIX Association, December 2004.

[42] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the fourteenth ACM Symposium on Operating systems principles (SOSP)*, pages 175–188. ACM Press, December 1993.

[43] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the fifteenth ACM Symposium on Operating systems principles (SOSP)*, pages 237–250. ACM Press, December 1995.

[44] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[45] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of the eighth IEEE workshop on Hot Topics in Operating Systems (HotOS)*, pages 15–20. IEEE Computer Society Press, May 2001.

[46] Daniel Mahrenholz, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Program instrumentation for debugging and monitoring with Aspect C. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 249–256. IEEE Computer Society, April 2002.

[47] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the thirteenth ACM Symposium on Operating systems principles (SOSP)*, pages 110–121. ACM Press, October 1991.

[48] Wolfgang Mauerer. *Linux Kernelarchitektur - Konzepte, Strukturen und Algorithmen von Kernel 2.6*. Carl Hanser Verlag, 1st edition, 2004.

[49] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[50] Thu D. Nguyen, Raj Varwani, and John Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In *Proceedings of the 1996 IPPS Workshop on Job scheduling strategies for parallel processing (IPPS)*, pages 155–174. Springer-Verlag, April 1996.

[51] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the third International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30. IEEE Computer Society, October 1982.

[52] Eric Parsons and Kenneth Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 24,1 of *ACM SIGMETRICS Performance Evaluation Review*, pages 57–67, New York, May 23–26 1996. ACM Press.

[53] 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 (SPPA)*, pages 237–245. ACM Press, July 1991.

[54] Margo I. Seltzer, Yasuhiro Endo, Small Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the second USENIX Symposium on Operating systems design and implementation (OSDI)*, pages 213–228. USENIX Association, October 1996.

[55] Kenneth C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the 1989 ACM conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 171–180. ACM Press, May 1989.

[56] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 106–126. Springer-Verlag, April 1995.

[57] David A. Solomon and Helen Custer. *Inside Windows NT*. Microsoft Press, 2nd edition, 1998.

[58] M. S. Squillante and E. D. Lazowska. Using processor cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[59] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the fourth USENIX Symposium on Operating systems design and implementation (OSDI)*, pages 117–130. USENIX Association, February 1999.

[60] The L4Ka Team. L4ka virtualization project webpage. `http://l4ka.org/projects/virtualization/`.

[61] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors. In *Proceedings of the 1993 ACM conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 272–274. ACM Press, June 1993.

[62] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the twelfth ACM Symposium on Operating systems principles (SOSP)*, pages 159–66. ACM Press, December 1989.

[63] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the third Virtual Machine Research and Technology Symposium*, pages 43–56. USENIX Association, May 6–7 2004.

[64] 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.

[65] Boris Weissman. Performance counters and state sharing annotations: A unified approach to thread locality. In *Proceedings of the eigth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 127–138. ACM Press, October 1998.

[66] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC2003)*, pages 15–21. ACM Press and IEEE Computer Society Press, April 2003.

[67] Robert W. Wisniewski and Luis F. Stevens. A model and tools for supporting parallel real-time applications in unix environments. In *Proceedings of the first IEEE Real Time Technology and Applications Symposium*, pages 126–133. IEEE Computer Society, May 1995.

[68] Karim Yaghmour and Mchel R. Dagenais. Measuring and characterizing system behaviour using kernel-event logging. In *Proceedings of the USENIX annual technical conference*, pages 13–26. USENIX Association, June 2000.

[69] Tom Zanussi, Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for transmitting data. In *Proceedings of the Linux Symposium*, pages 494–507, July 2003.

[70] Songnian Zhou and Timothy Brecht. Processor-pool-based scheduling for large-scale numa multiprocessors. In *Proceedings of the 1991 ACM conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 133–142. ACM Press, May 1991.